FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Store-and-forward CDC packet transmission in digital systems

Tiago Filipe Almeida Campos

Master in Electrical and Computers Engineering

Supervisor: José Carlos Alves

Co-supervisor: Manuel Oliveira

Co-supervisor: Miguel Falcão Sousa

July 16, 2019

Abstract

In order to reduce power utilisation or allow higher propagation delay between flip-flops, modern integrated circuits are composed of various sub-circuits operating at different clock frequencies. The resulting clock signals delimit "clock domains", which are the regions of the circuit they affect.

When creating these integrated circuits, the transmission of data between the clock domains is a significant point of interest by systems designers. Limited bandwidth in the transmissions can be responsible for a processing power bottleneck that affects the entire system if, for example, functional blocks halt their operation while waiting for communications to be processed by other blocks.

Transferring signals between clock domains is known as clock domain crossing (CDC). Clock domain crossing is inherently expensive in terms of area and latency as it requires overcoming issues related to the physical nature of integrated circuit latches, in particular metastability. Metastable behaviour is difficult to analyse as it does not manifest in register-transfer level (RTL) simulation.

Due to the difficulty of analysing metastability, and in order to ensure that a product will work, designers often opt for generic data synchronisation solutions that are not entirely suited to the nature of the data being transferred. These generic solutions often equate to sub-optimal results in both area and performance. These inefficiencies can be mitigated through the development of clock synchronisation mechanisms that provide functional abstraction on top of the domain-crossing data.

This dissertation presents a new clock domain crossing mechanism that allows two clock domains to share a common random access memory (RAM) to transfer packet-based data. The mechanism consists of a memory controller that coordinates commands from a push (write) domain and a pop (read) domain.

During the design and development of the memory controller, focus lies in the study and implementation of efficient synchronisation structures. Two of the primary goals are to cause minimum performance overhead, and to eliminate the need for separate synchronisation and packet storage memories. In essence, the controller is an extension of an asynchronous first-in-first-out (FIFO) controller with added functionality, supporting multiple virtual FIFOs and variable-length data packets. Additionally, it allows the pop domain to read packets in order even if they were transmitted out of order.

Keywords— clock domain crossing, digital systems, metastability, segmented buffer, synchronisers

Resumo

De forma a minimizar consumo energético ou permitir atrasos de propagação mais elevados entre *flip-flops*, circuitos integrados modernos são compostos por vários sub-circuitos a trabalhar a diferentes frequências. Os sinais de relógio resultantes delimitam "domínios de relógio", sendo estes regiões do circuito que os mesmos afetam.

Durante o desenvolvimento destes circuitos integrados, a transmissão de dados entre os domínios de relógio é um dos principais focos de atenção por parte de projetistas de sistemas digitais. Largura de banda reduzida nestas transmissões pode resultar numa limitação do poder de processamento do sistema como um todo, por exemplo se alguns blocos funcionais interrompem a sua própria atividade enquanto esperam que dados sejam processados por outros blocos.

Sincronização de sinais entre domínios de relógio é inerentemente custosa em termos de área e latência devido à necessidade de superar problemas de natureza física dos *latches* utilizados em circuitos integrados, nomeadamente metaestabilidade. Estes problemas não se manifestam em simulação *register-transfer level* (RTL), o que leva os projetistas a utilizar mecanismos genéricos de sincronização de dados que podem não ser os ideais considerando a natureza dos dados a transferir.

Estes mecanismos genéricos de sincronização tipicamente levam a problemas de eficiência em termos de desempenho e de utilização de área. Estas ineficiências podem ser resolvidas através do desenvolvimento de novos mecanismos de sincronização que providenciem funcionalidades mais complexas como suplemento à sincronização de dados propriamente dita.

Esta dissertação apresenta um mecanismo de sincronização que permite que dois domínios de relógio partilhem uma memória *random access* (RAM) comum, sendo esta uma plataforma de retenção e transferência de dados organizados por pacotes. O mecanismo consiste num controlador de memória que coordena comandos provenientes de um domínio de *push* (escrita) e um domínio de *pop* (leitura).

No decorrer do desenvolvimento do controlador de memória, o foco situar-se-á no estudo e implementação de estruturas de sincronização eficientes. Tem-se como principais objetivos a redução do impacto de desempenho causado pelo mecanismo de sincronização, e a eliminação da necessidade de existência de memórias separadas para sincronização e para armazenamento de pacotes. Em essência, o controlador é uma extensão de um controlador de memória *first-in-first-out* (FIFO) assíncrona, adicionando suporte para múltiplas FIFOs virtuais, pacotes de tamanho variável e possibilidade de re-ordenação de pacotes.

Palavras-chave— buffer segmentado, metaestabilidade, sincronizadores, sistemas digitais

Agradecimentos

Queria agradecer ao meu orientador, Prof. José Carlos Alves, por todo o apoio prestado e por todo esforço em guiar este trabalho, esforço que certamente superou todas as expectativas.

Aos meus orientadores na Synopsys Porto, Manuel Oliveira e Miguel Falcão, pela atitude extremamente profissional e proativa, estando sempre dispostos a ajudar inconsequente de imensas outras responsabilidades.

Agradeço também a todos os restantes membros da equipa PCIe Synopsys Porto, que sempre demonstraram imenso interesse e vontade de ajudar. Thank you Ali Azarian, for the effort you put into reviewing the document!

À minha família, por fazer isto tudo possível. Em especial à minha irmã Mónica por ter tido sempre os meus interesses em primeiro plano, à minha mãe Conceição e avó Isabel.

A todos os colegas de curso que traçaram este percurso comigo durante estes cinco anos, em especial Bruno, Samuel, Afonso e Luís. A todos os amigos da "terrinha" que me acompanharam durante estes anos, Francisco, José, Hugo e restantes frequentadores do "dunas".

Tiago

“ What lies behind us and what lies before us are
tiny matters compared to what lies within us. ”

Ralph Waldo Emerson

Contents

List of Figures

List of Tables

Abbreviations and Symbols

ACK	Acknowledge
addr	Address
algn	Align
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application-Specific Integrated Circuit
CDC	Clock Domain Crossing
ctr	Counter
dp	Depth
DPRAM	Dual-Port Random Access Memory
DUT	Design Under Test
FET	Field-Effect Transistor
FF	Flip-Flop
FIFO	First-In-First-Out
fwd	Forward
ID	Identifier
MSb	Most Significant Bit
MTBF	Mean Time Between Failures
MTU	Maximum Transmission Unit
MUX	Multiplexer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
ptr	Pointer
RAM	Random Access Memory
REQ	Request
RTL	Register Transfer Level
SBC	Segmented Buffer Controller
seg	Segment
SoC	System-on-Chip
SysML	Systems Modelling Language
TSMC	Taiwan Semiconductor Manufacturing Company
wd	Width

Chapter 1

Introduction

Chapter Outline

DIGITAL systems have been at the forefront of technological evolution in recent years, being responsible for significant advancements to human quality-of-life. Semiconductor manufacturing processes have evolved at an exponential rate, modelled by "Moore's Law" [23], which states that the number of transistors in an integrated circuit doubles approximately every two years.

As transistor density increases, the amount of logic that can fit in an integrated circuit increases accordingly, allowing integrated circuits to perform more and more complex operations in extremely small packages.

As integrated circuits evolve in processing capacity, they become feasible for integration in new markets. The mobile communications market, especially with the advent of smartphones, is an example where a race for processing power is critical in order to provide products that more consumers will find appealing. The rapid growth of the mobile communications market has propagated to growth in other markets, for example internet infrastructure and services, which rely heavily on large server farms and thus creating a loop on integrated circuit reliance. Parallely, the automotive industry has also seen an increase in integrated circuit utilisation across all functionality of their products, from engine management and stability control to infotainment.

In the mentioned examples, industry market capitalisation is defined by the functionality that the competing companies can fit into their integrated circuit solutions. Furthermore, these functionalities must also comply to engineering restrictions such as maximum power utilisation, acceptable system performance, and reliability.

To fulfil this large amount of requirements, digital systems engineers typically develop a System-on-Chip (SoC). This SoC is an immensely complex application-specific integrated

circuit (ASIC) that combines circuits developed simultaneously by different teams in different companies.

1.1 Motivation and Problem Overview

Modern integrated circuit development, particularly of SoCs, consists in the integration of various functional blocks developed simultaneously by different teams. As power utilisation must be kept to a minimum, each functional block is designed to work at the minimum clock frequency that meets its design requirements.

Integration of various functional blocks leads to the existence of multiple clock domains operating in parallel inside a single chip. A clock domain is any region of the circuit that operates under a particular clock signal (i.e., with the same frequency and phase).

Communication between these clock domains is the source of various design problems. Although there are standard specifications for SoC interconnect (an example being the Advanced Microcontroller Bus Architecture, AMBA) that allow the blocks to communicate on the same terms, they must be designed in order to take full advantage of the buses' communication performance capabilities.

A block with poorly designed domain-crossing communications can cause needless performance degradation on other blocks if it cannot accept all incoming data quickly enough, causing a cascade of system halting as other blocks wait for the data to be read.

This document will focus on the communications that take place between clock domains. Our focus will be to develop a communication mechanism that allows two blocks, working under different clock domains, to streamline their data communications and maximise system performance. For this, we associate the following goals to the domain-crossing data transmission:

- Provide simple data semantics: Provide a small abstraction layer on top of the raw data that allows more complex transactions
- Minimise latency and area: Use a common memory as a data storage and transmission platform
- Maximise performance: Allow both domains to work at their maximum speed and allow asynchronous processing and out-of-order data arrival

The given list of goals can be re-written as a set of questions which will be considered throughout the document:

- **Question 1.** How to map data transactions between functional blocks into a simple structure that facilitates asynchronous processing?
- **Question 2.** How to map this data structure into a physical memory?

- **Question 3.** How to allow this physical memory to be safely shared by two clock domains?

- **Question 4.** How to do this while maximising performance?

Although our focus is on the domain-crossing part of the system, we must understand how the data transmitted between domains is structured. As such, it will be described as part of the problem. **Questions 1 and 2** serve to define the data transmission context, while **questions 3 and 4**, which focus on the clock-domain crossing aspects, are the main focus of the dissertation.

Variable-length packets are the base of a preliminary answer to **question 1**. The transmission is based upon the data structure shown in figure 1.1, which introduces many of the critical concepts that should be understood throughout the document.

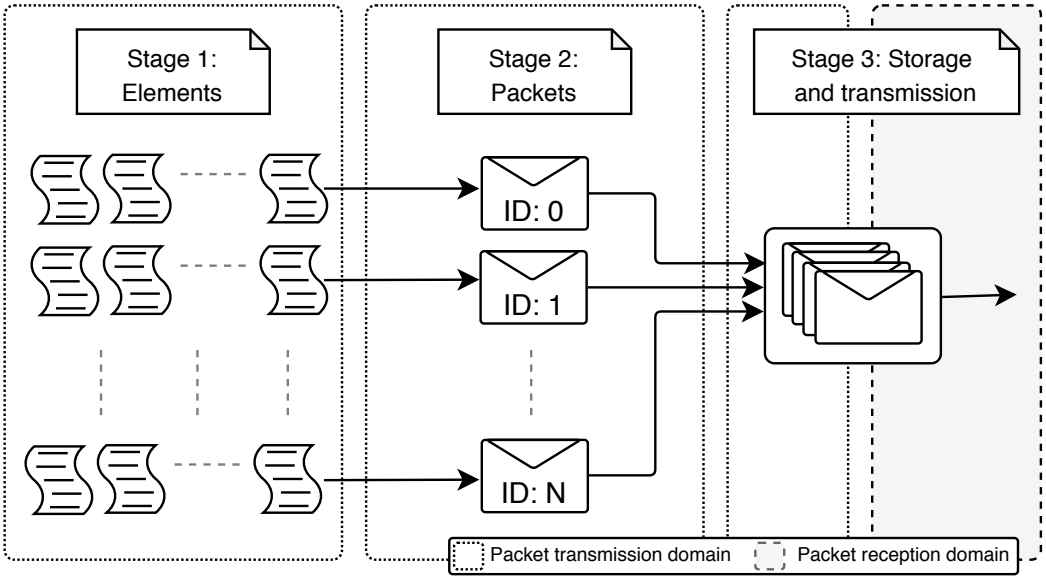


Figure 1.1: Block diagram of element composition into packets and their storage

In the given data structure, the atomic transmission unit is the element. In order to support variable-length data transmissions, the concept of packet is used. A packet is composed of one or more related elements. As these packets may be received out of order, each packet is linked to a unique identifier (ID), allowing the domains to know if packets are out-of-order and thus facilitating re-ordering logic between packets (not between elements). The work described in this dissertation will focus on the third stage, packet storage and transmission.

Figure 1.2 shows a simple application of this data transmission structure. A Peripheral Component Interconnect (PCI) device listens for incoming requests with an associated request ID. The PCI device processes and responds to the request with a completion, associated with the same ID as the previously received request. In the presented packet data structure, each completion is a packet.

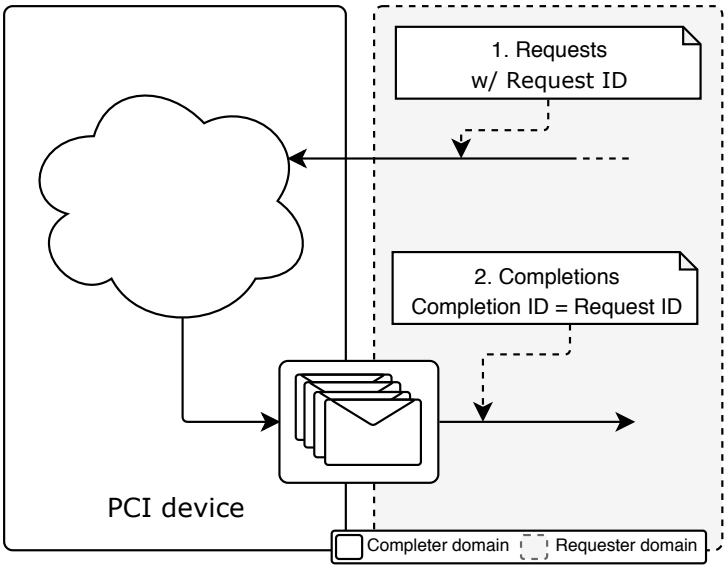


Figure 1.2: Block diagram of store-&-forward packet transmission in a PCI device

The transmission side benefits from having more complex control logic. For example, an internal device error can cause invalidation of a completion that is being written (i.e., that contains some but not all elements that constitute the packet). In this case, dropping the entire packet from memory is desired.

For **question 2**, in order to map this packet storage into a physical memory, we require a memory structure that keeps track of the status of each packet. Typically, this involves segregating a single memory into multiple segments, each segment pertaining to a specific packet. This memory structure is known as a segmented buffer.

Figure 1.3 shows the structure of a typical segmented buffer. These segmented buffers are usually single-clocked designs that abstract the RAM address space into independent segments. Each segment is written to and read from as a FIFO and can be used to store one packet, matching the packet ID with the segment ID.

The represented memory layout shows each segment as a contiguous section of memory. Since different RAM address space abstraction techniques can be used, it is not necessary for it to be contiguous. In this layout, however, since each segment has defined start and end addresses, the segment depth must be limited by setting a maximum packet size. This maximum packet size is known as the maximum transmission unit (MTU).

In data transmission between domains, a single-clocked segmented buffer can be used by the transmission and reception side to store outgoing and incoming data, respectively. Data synchronisation must be handled externally through a generic data synchroniser, which is not optimal as it usually requires choosing between either a significant memory overhead or performance degradation.

A dual-port RAM (DPRAM) could be used to write and read data directly by different clock

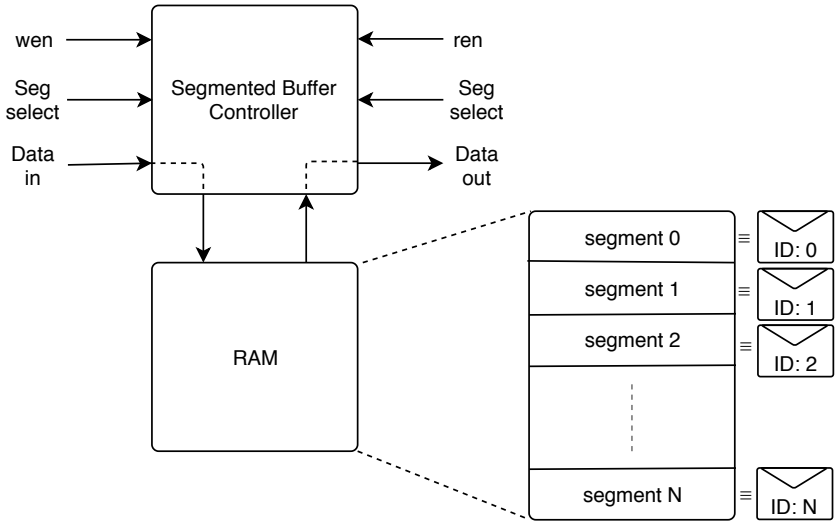


Figure 1.3: Block diagram of a typical segmented buffer interface

domains. However, the segmented buffer controller must be adapted to support this change. This adaptation, consisting in the decomposition of the segmented buffer controller (SBC) into two isolated clock domains and the synchronisation mechanisms between them, is the core of this work that aims to explore **questions 3 and 4**. For this, we will start by reviewing the problems related to clock domain crossing.

1.2 Contibutions

In this dissertation we aim to extend the functionality of a segmented buffer controller to allow asynchronous usage of the buffer by two independent clock domains.

Implementing this segmented buffer packet transmission functionality across clock domains includes the work of separating the segmented buffer controller into two input-output interfaces, each synchronous one of the clocks.

The main focus of the dissertation is in the choice of the internal synchronisation mechanisms required to make the segmented buffer work simultaneously in both domains. As such, a review of current clock domain crossing methodology is required.

The final goal of the dissertation is the introduction of a new clock domain crossing (CDC) synchronisation mechanism. The mechanism allows two clock domains to share a common memory to transmit data that is structured into packets. It is innovative in that it allows the transmission domain to stream data continuously, possibly out of order, and only confirm it later. The reception domain can choose to read the packets in the order it desires. Furthermore, it allows both domains to use the memory at each domains' maximum speed (i.e., the transmission bandwidth is not lowered by the required CDC synchronisation structures).

1.3 Organisation

This dissertation is composed of five chapters and one appendix. The document chapters are structured as follows:

- **Chapter 1 - Introduction** introduces the fundamental concepts of the problem and explains how a segmented buffer can be used to solve it for communication across blocks in single-clocked designs.
- **Chapter 2 - Background and Previous Work** provides a review on the state of the art of clock domain crossing, presenting the main problems related to it and current solutions.
- **Chapter 3 - Approach** describes an intermediate stage between planning and implementation. A generic top-level architecture for the module is introduced with black-boxed synchronisation blocks. Afterwards, some synchronisation block candidates are detailed and evaluated through simulation and synthesis measurements.
- **Chapter 4 - Implementation and Results** details the chosen synchronisation architectures for the implemented module, which consists of a segmented buffer with integrated CDC functionality. An existing generic solution consisting of a single-clocked segmented buffer in series with an auxiliary generic data synchroniser is also presented to provide a basis of comparison in terms of area and performance.
- **Chapter 5 - Conclusions and Future Work** concludes by presenting remarks and a summary of the work performed, further detailing with suggestions for future work.

Chapter 2

Background and Previous Work

Chapter Outline

CLOCK domain crossing issues have been the subject of many research efforts. These efforts span in scope from the definition of the associated physical [3, 25, 32] and functional [9, 26, 31] problems to practical designs that overcome them [4, 5, 6, 8, 13, 27] and verification [14, 17, 18, 19, 20] of said designs.

The main issue behind clock domain crossing is metastability. Metastability is caused by the physical properties of integrated circuit latches and in turn, may cause other functional issues, including data incoherency and data loss or repetition. Data loss or repetition is not an exclusively metastability-bound problem may also be caused by faulty synchronous logic on either domain. If the reception or transmission domain synchronous logic does not account for the possibility that the clock domains might be working at a significantly different speed, data loss or repetition can also occur.

In practice, clock domain crossing problems are solved by employing synchronisation mechanisms known as synchronisers, as described in section 2.2. Although these mechanisms do allow the design to overcome metastability, they are vulnerable to erroneous employment, effectively rendering the synchroniser useless. In order to provide an understanding of why and how synchronisers are used, this chapter summarises the concerns behind metastability, synchronisation mechanisms that overcome its issues, and their usage restrictions.

2.1 CDC Background

This section reviews the problems related to clock domain crossing, namely metastability, data loss, data repetition, CDC jitter and data incoherency.

2.1.1 Metastability

Metastability is a state where a sampled digital signal holds an undefined value, neither 0 nor 1, instead holding an intermediate voltage value that will eventually decay into a defined logic level. Metastability occurs when a signal generated in one clock domain changes too close to the rising edge of a clock signal on another clock domain, causing a violation of setup or hold times of the sampling flip-flops [20].

By definition, metastable behaviour is unpredictable. The time it takes to decay and its final logic value are unknown, although the time it takes to decay can be modelled by a negative exponential probability distribution [9]. As it cannot be avoided, the only option to overcome metastability is to mitigate its propagation through the circuit so that the probability of it passing through combinational logic is low enough that it is unlikely for metastability to manifest in the lifetime of the circuit.

Figure 2.1 highlights the issue with a representation of the input and output waveforms of a D-type flip-flop affected by metastability. Metastability failure occurs when this metastable value reaches combinational parts of the circuit, allowing the propagation of an undefined logic value.

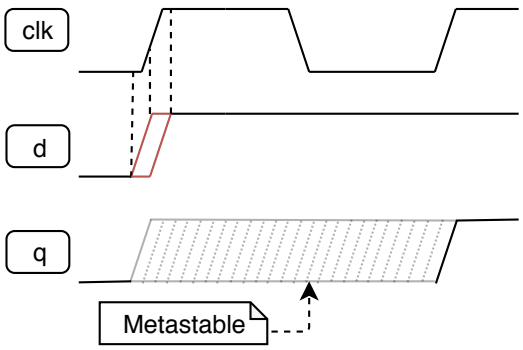


Figure 2.1: Waveforms of metastability in a D flip-flop

The susceptibility of a latch to metastability is technology dependent. Typically, smaller latches are more resilient to it as they are capable of recovering faster [3]. A window of metastability failure can be determined through equation 2.1 [25]. This is the time window (i.e., the time difference between the sampling clock and input data edges) in which the latch will not be able to resolve the metastability before it reaches the remainder of the circuit.

$$\delta(t_r) = T_0 e^{t_r/\tau} \quad [\mathrm{s}] \quad (2.1)$$

In equation 2.1, t_r is the maximum allowed metastability resolution time. Assuming that a typical D flip-flop is comprised of two latches in series, one triggered on the positive edge and the other on the negative edge, the maximum allowed resolution time for the first latch of a flip-flop is usually half a clock cycle, since the following latch is triggered on the complementary clock edge.

The window of susceptibility to metastability, T_0, dictates the maximum phase between data and sampling clock edges where metastability resolution time is non-zero. In turn, τ is the metastability regeneration time constant, which dictates how quickly the latch is capable of driving the output voltage away from the metastable level.

The variables τ and T_0 are dependent on the physical latch characteristics, which in turn are dependant on technology, latch dimensions and fan-out. However, on purely digital designs, latch dimensions and fan-out do not tend to be significant [25].

From this metastability failure window, we can deduce a "Mean-Time Between Failures" (MTBF) statistic only additionally requiring the source data frequency (f_d) and the destination clock frequency (f_c) variables.

$$MTBF(t_r) = \frac{e^{t_r/\tau}}{f_d f_c T_0} \quad [\text{s}] \quad (2.2)$$

Equation 2.2 shows the classic MTBF formula [8, 20]. The f_d and f_c variables show what affects the probability of metastability occurring. For example, the likelihood of metastability increases linearly with the data rate and destination clock frequency (although in practice it scales quadratically with the destination frequency since t_r is typically a function of $1/f_c$).

The f_d variable introduces a concept which may not be immediately evident. It does not necessarily represent the source clock frequency as it may be fully asynchronous. It is instead the maximum frequency of the input data. For example, if the data only changes once every two source clock cycles, then f_d can be safely defined as half the frequency of the source domain.

This discrepancy between the source clock and data frequencies can result in dangerous miscalculations of MTBF. In particular, combinational circuits may cause glitching in the CDC path, which highly increases the number of transitions in the signal, increasing the likelihood of metastability by unintentionally increasing f_d [26]. Therefore, it is required to register all signals at the exit of the transmission clock domain so that the CDC signal is not affected by glitching.

Avoiding metastability involves designing circuits that increase the MTBF of all clock domain crossing paths up to a desired target. These circuits are known as synchronisers.

2.1.2 Data Loss, Repetition and CDC jitter

As long as each transition of the source signal is captured on the destination clock domain, data is not lost. This highlights the cases where data loss may exist: if synchronising into a slower clock domain, we must ensure that the source clock domain does not change the data for a

long enough time to guarantee that the destination clock domain can sample it. In contrast, if sampling into a faster clock domain, data repetition may occur, which consists in the same bit value being sampled multiple times by the faster domain. In this case, it is necessary to implement a mechanism that ensures the receiving clock domain knows when to recognise a new bit. Data is considered to have integrity if it does not suffer from loss or repetition [20].

Metastability can cause the unpredictable introduction of data loss and repetition issues. As shown in figure 2.2, signal *d1*, synchronous to the *clk1* domain, is sampled into a faster clock domain, *clk2*, which is approximately twice as fast. The *d1* signal in the source domain follows the pattern 0101, being reasonable to expect that it will follow 00110011 in the reception domain. However if, for example, the first transition of *d1* goes metastable, the resulting sampled bit has an unknown value and we may sample either 00010011 or 00110011. This issue is present for all clock frequency ratios and is known as CDC jitter.

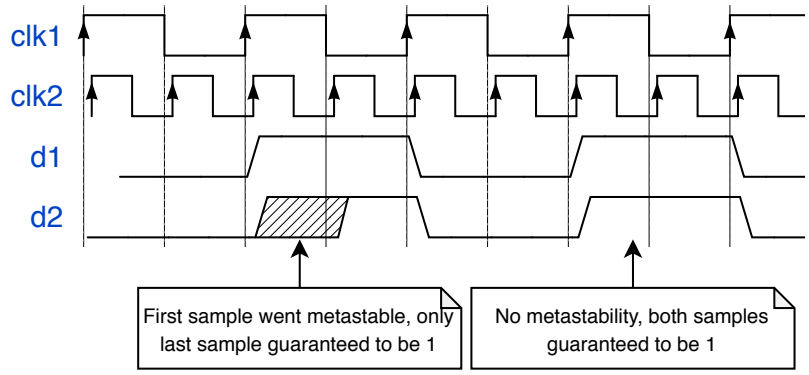


Figure 2.2: Waveforms of CDC jitter caused by metastability

2.1.3 Data Convergence Incoherency

The issues in sections 2.1.1 and 2.1.2 were described for single-bit signals. For multi-bit signals, the same issues arise in each of the bits. This is especially troublesome in the case of CDC jitter. If one of the bits synchronises on the first clock edge, and another on the second clock edge, data is *incoherent* during that first clock cycle, since it contains values sampled from two different clock cycles of the transmitting domain. This means that, typically, we cannot merely replicate single-bit synchronisation mechanisms for each bit of a multiple-bit signal, as the synchronised signal could suffer from incoherency, and more complex synchronisation methods must be employed [20].

2.2 Current Synchronisation Approaches

Synchronisers are circuits designed to overcome the clock domain crossing issues shown in section 2.1. These synchronisers can be characterised by tradeoffs in area, latency, throughput and implementation difficulty.

The simplest synchroniser is the two-flip-flop (two-FF) synchroniser shown in figure 2.3, consisting of two destination-clocked flip-flops in series. As the smallest and most basic synchroniser, it is often used as a building-block for more complex synchronisers and therefore is referred to as "fundamental synchroniser". The more complex synchronisers overcome limitations of the fundamental synchroniser, such as being vulnerable to data incoherency, data loss and data repetition.

We will focus on the most common synchroniser types that support any arbitrary clock frequency ratios although some niche designs exist for specific cases, e.g. domains that are derivations of the same main clock or domains that only differ in phase and not frequency. All shown synchronisers implement unidirectional data transmission as bidirectional synchronisation consists of the mirrored replication of these synchronisers.

2.2.1 Fundamental Synchroniser

As previously mentioned, the simplest and most common synchroniser is the two-FF synchroniser [5], as shown in figure 2.3. This is a single-bit synchroniser that only contains metastability and does not protect against data coherency and integrity issues.

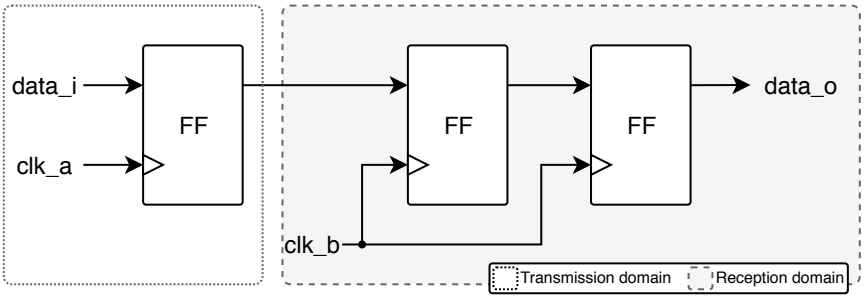


Figure 2.3: Schematic of a two-FF synchroniser

The first flip-flop samples the asynchronous input signal into the new clock domain and waits for a full clock cycle to allow any metastability on the first stage output to decay, then the first stage output is sampled again by the same clock into a second stage flip-flop, with the goal that the second stage signal is now a stable and valid signal in the new clock domain.

The two-FF synchroniser can be extended both in data width and pipeline stages, resulting in a N x M matrix of flip-flops as shown in figure 2.4. Increasing the number of pipeline stages (N) increases the MTBF according to section 2.2.1.2. Increasing the data width (M) allows passing multi-bit signals across domains. However, this increase in data width must be

supplemented by techniques that ensure that no data incoherency occurs. One such technique is Gray-coding the data bus.

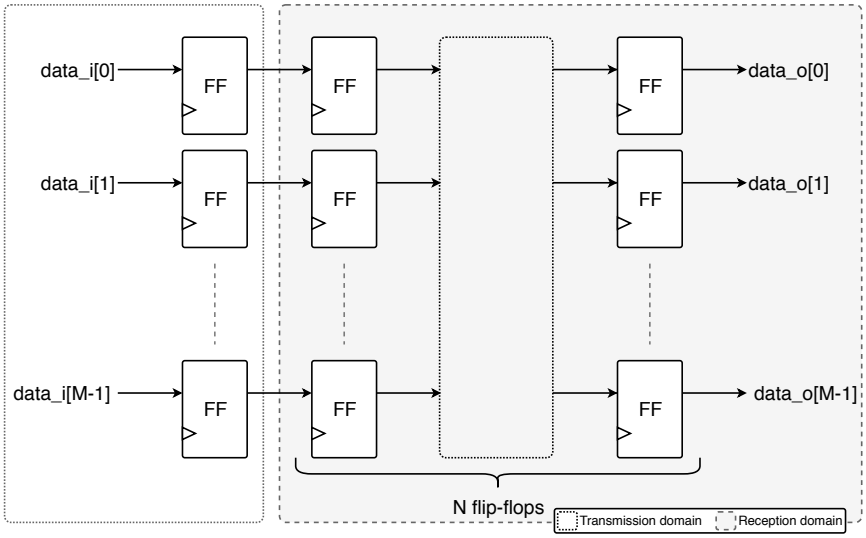


Figure 2.4: Block diagram of the fundamental synchroniser as a N x M matrix

2.2.1.1 Gray Coding

Gray code is a binary numeral system where two consecutive values differ only in one bit. This numeral system is compared with decimal and classic binary in figure 2.5 [5]. If using Gray codification, multi-bit signals such as counters that only change in sequence (i.e., where their value only increases or decreases by a maximum of 1 each source clock cycle) can be synchronised by the fundamental synchroniser without being vulnerable to data incoherency.

This Gray-coded bus is not vulnerable to data incoherency because at any one point metastability can only occur in one of the bits of the signal, in which case it does not matter to which logical value the metastability resolves to as both are valid.

Gray coding is typically used to synchronise counters and read/write pointers which can, in turn, be used to create CDC mechanisms for data that cannot follow to the restrictions of Gray encoding.

2.2.1.2 MTBF of a Fundamental Synchroniser

As data and sampling frequencies increase, the CDC MTBF decreases, which must be compensated by increasing the number of pipeline stages. Due to this, very high-frequency circuits may require 3 or more stages. This section presents a quick review of how to calculate the MTBF of a fundamental synchroniser.

Physical characterisation of flip-flop and latch MTBF has been a topic of extensive research [7, 12, 15, 16]. Equation 2.2 was extended to support chained latches by Jones et al. [13].

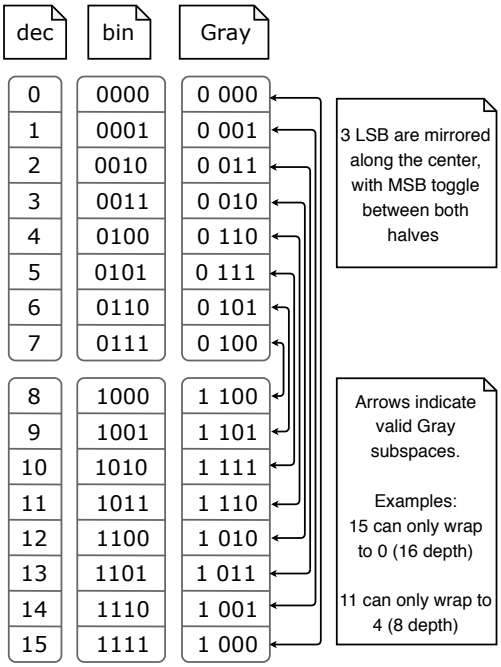


Figure 2.5: Gray coding table for a 4-bit sequence

This introduces the concept of data arrival window where, for a particular latch at stage i, the window of time between the two data arrival times that cause a metastability resolution time equal to t_r is shown in equation 2.3.

The data arrival window can be used to calculate the MTBF of a new latch appended to a path with a specific MTBF. If chaining stage $i+1$ after stage i, MTBF increases according to equation 2.4.

Assuming all stages have the same t_r, τ and T_0 we may equate the MTBF for a chain of N latches we can simplify to equation 2.5.

This gives us the MTBF for a single bit. If the data is not comprised of a single bit, the aggregated MTBF of all bits must be calculated according to equation 2.6.

$$\Delta tin_i(t_{r_i}) = \frac{T_{0_i}}{e^{t_{r_i}/\tau_i}} \tag{2.3}$$

$$MTBF_{i+1} = MTBF_i \cdot \frac{\tau_{i+1}}{\Delta tin_{i+1}(t_{r_{i+1}})} \tag{2.4}$$

$$MTBF(t_r, N) = \left(\frac{e^{t_r/\tau}}{T_0}\right)^N \cdot \frac{\tau^{N-1}}{f_c f_d} \tag{2.5}$$

$$MTBF_{agrt} = \frac{1}{\sum_{i=0}^{M} \frac{1}{MTBF_i}} \tag{2.6}$$

Through equations 2.5 and 2.6, it is possible to obtain an estimation of the MTBF of a bus synchronised through fundamental synchronisers. It is important to note that these equations present only a very rough estimation of the MTBF. In particular, equation 2.5 already introduces a large approximation by assuming all latches in a fundamental synchroniser have the same τ and T_0, where the two latches in a flip-flop typically do have different physical characteristics. Secondly, equation 2.6 assumes all wires in the bus have the same MTBF. However, since the purpose of this overview is to get a rough estimation of MTBF, these approximations will be considered acceptable.

2.2.2 MUX Synchroniser

The multiplexer (MUX) synchroniser shown in figure 2.6, also known as MUX recirculation synchroniser, is a multi-bit synchroniser that attempts to solve the data incoherency issue by synchronising a single control bit which asserts that the asynchronous multi-bit data is stable and ready to be sampled [26]. This requires control logic in the transmitting domain in order to generate the control signal and block data transitions while the receiving domain samples the signal. The control logic must be designed carefully in order to compensate for the latency added by the fundamental synchroniser.

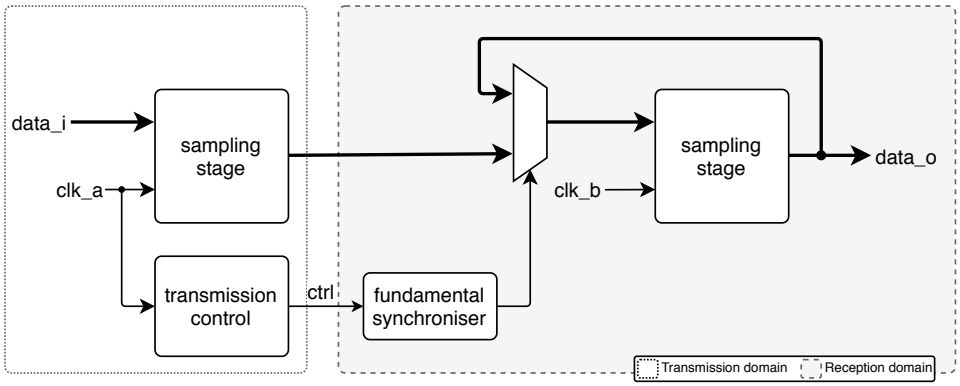


Figure 2.6: Block diagram of a MUX synchroniser

This synchroniser turns multi-bit synchronisation into a single-bit synchronisation problem despite not solving the issue that data may be lost if the clock frequency ratios are unknown. To solve this, a full handshake protocol may be implemented at the cost of extra latency, as shown in section 2.2.3.

2.2.3 Handshake Synchroniser

The handshake synchroniser [21] shown in figure 2.7 is a multi-bit synchroniser similar to the MUX synchroniser in that it synchronises single-bit control signals to ensure that a multi-bit signal can be safely sampled.

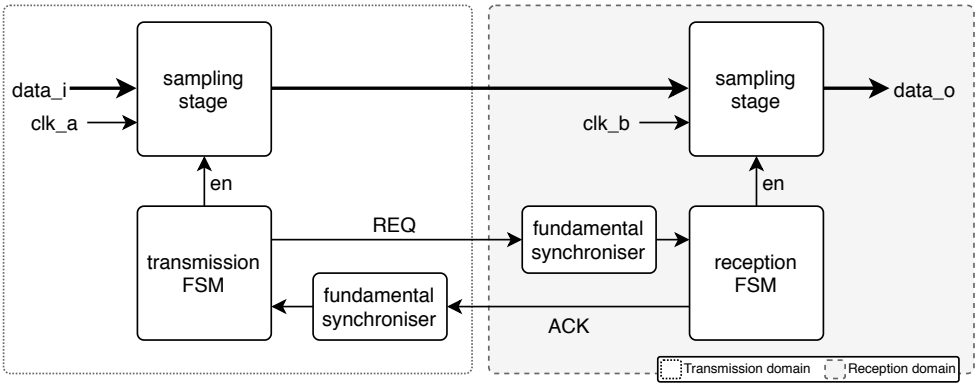


Figure 2.7: Block diagram of a handshake synchroniser

The main difference between the MUX and the handshake synchronisers is that the handshake synchroniser implements a feedback signal from the receiving to the transmitting domain that enables closed-loop transmission control. The two control bits, one in each direction, are normally typically to as request (REQ) and acknowledge (ACK) [6].

The synchroniser can be designed with either a push or a pull interface depending on which side initiates the transmission. On a push interface, the transmitter asserts REQ to write on the receiver which then asserts ACK. On a pull interface, the receiver asserts REQ to request data and the transmitter then asserts ACK when the data is valid. A push interface is shown in figure 2.7.

To use this synchroniser, it is required that both the transmitter and receiver implement state machines to manage REQ and ACK. The main disadvantage of the synchroniser is the latency created by this request-acknowledge mechanism. The mechanism creates a latency between transmissions of two fundamental synchronisers plus the state machine delay of each domain. The transmitting domain must hold the data stable between these transmissions, resulting in low throughput.

2.2.4 Asynchronous FIFO

A typical asynchronous first-in-first-out (FIFO) synchroniser is shown in figure 2.8 [4]. In this configuration, the clock domains use a dual-port memory as a middleman for communication. The main advantage with the use of the memory comes from its ability to allow data to accumulate in the case it is being written faster than it is read. This possibility of accumulation eliminates the need to wait for the other clock domain when reading or writing, allowing both domains to work independently and at their maximum speed while maintaining data integrity.

The read and write addresses generated to access the FIFO memory are synchronised between domains. The addresses are Gray encoded in order to allow usage of fundamental synchronisers without data incoherency issues. The addresses are synchronised so that the

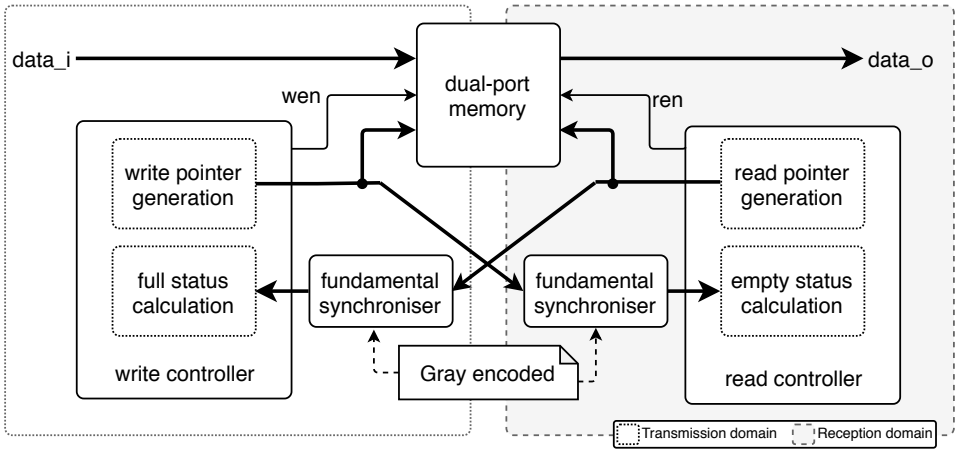


Figure 2.8: Block diagram of an asynchronous FIFO synchroniser

reading domain may know when the FIFO is empty, and the writing domain when the FIFO is full.

This synchroniser is particularly interesting as it implements some functionality that is useful from a CDC store-and-forward perspective, namely sharing a common memory with surrounding synchronisation to close the control loop.

2.2.4.1 FIFO Synchronisation Logic

To understand this memory sharing mechanism, let's take a closer look on how and why access pointer synchronisation is used to implement this synchroniser.

In a CDC architecture, the synchronised data is always delayed in relation to the "real" data. That is, the reception sides' perception of the synchronised data is a delayed copy of it. This is because the synchronisation mechanism itself introduces latency, which is further worsened by CDC jitter.

The FIFO architecture must be designed accounting for this. Each clock domain must assume that the worst-case scenario has happened since the last synchronisation, and re-synchronisations allow lifting this assumption.

In asynchronous FIFOs, the worst-case scenario is actually where the other domain has idled since the last synchronisation. For example, the push domain worst-case scenario is when the memory is full. The case where the memory fills quickest is when the pop domain is not reading any data. Therefore, this is what the push domain must assume until the synchronised data refreshes. The opposite case is also true for the pop domain, where it must assume the push domain has not written any data unless it has explicitly received information confirming so. These pessimistic assumptions are known as CDC pessimism.

The most commonly used synchronisation data is the read and write pointers of each domain into the other. This allows each side to keep track of the other sides' activity and use it to lift full and empty status flags. The pointers are already generated in each domain for

memory access logic, and it is not crucial for the receiving domain to catch every transition of the pointer, making this choice robust against arbitrary clock frequency ratios.

As it is synchronous to the write pointer, the push domain knows the exact clock cycle where each data write happens, and it also has access to the last synchronised read pointer. The FIFO fills when the write pointer, which is always ahead of the read pointer, catches up to the synchronised read pointer which means it has wrapped around the FIFO and can no longer write data without overwriting data which has still not been read. Due to CDC pessimism, this data may have already been read although the push domain has no knowledge of this yet.

Consider the situation where the FIFO is full (e.g. rptr $= 0$, wptr $= 0$) and the pop domain reads until empty before the next synchronisation of the read pointers to the push domain. After re-synchronisation, the actual FIFO state is empty but the push domain has seen no change in the read pointer, as illustrated in figure 2.9.

The push side now erroneously thinks the FIFO is full while in reality it is empty. Both sides halt activity, stalling the system. This can be avoided by having an extra bit that encodes the wrap state of the pointers. This bit toggles each time the pointer wraps around from the end to the start of the FIFO. This bit can just be concatenated with the real read and write pointers as their most-significant bit (MSb), making this encoding trivial in terms of pointer arithmetics. Figure 2.10 shows the same situation with the wrap bit added, which allows for solving this issue.

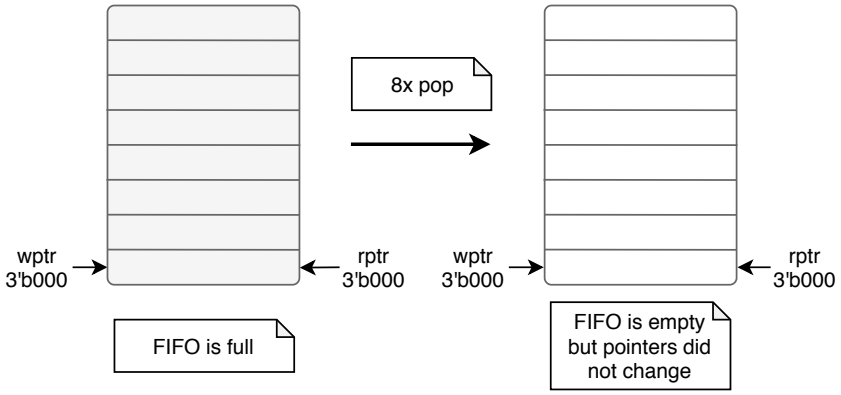


Figure 2.9: RAM status showing issue in CDC FIFO status flag calculation

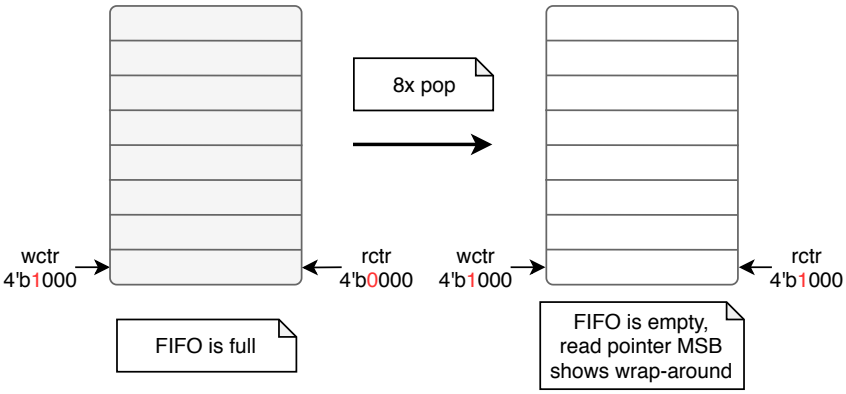


Figure 2.10: RAM status in CDC FIFO status flag calculation with added wrap-status bit

2.3 Summary of Synchronisers

Table 2.1 shows a summarised comparison of the main characteristics and tradeoffs of each synchroniser. This table is only intended to represent a rough comparison as implementation details may affect these characteristics.

Data frequency represents the maximum rate at which the synchronised data can change and can be seen as the inverse of the minimum latency between consecutive transmissions. The data frequency is high for the fundamental synchronisers as they are open loop, although this high data frequency is not necessarily useful without data integrity and coherency.

Data integrity represents protection against data loss and data repetition. It ensures that the receiving domain catches each transmitting domain change once and only once. The MUX synchroniser only partially supports this, as the control logic must be designed manually according to the clock frequency ratios.

Table 2.1: Summary of studied synchronisers

	Data freq.^{cy}	Data integ.^{rty}	Data coh.^{ncy}	Area	Implementation
Fund.^{tal}	High	No	No	Small	Very easy
Fund.^{tal} (Gray)	High	No	Yes	Medium	Easy
MUX	Medium	Partial	Yes*	Medium	Moderate
Handshake	Low	Yes	Yes	Medium	Moderate
FIFO	High	Yes	Yes	Large	Difficult

Data coherency is the lack of the data incoherency problem. Only the non-Gray-encoded fundamental synchroniser is vulnerable to incoherency. The Gray-encoded synchroniser does not lose coherency regardless of any metastable bit resolution outcome. The other synchronisers do not lose coherency by instead guaranteeing that the data is stable at the moment it is sampled. In the MUX synchroniser, this also implies some manual implementation that considers clock frequency ratios.

Area accounts for the circuit area required to perform the synchronisation. The fundamental synchroniser does not have any overhead as it is open-loop and requires no extra logic. The Gray synchroniser requires binary-encoded data to be converted into Gray-encoded data, although this conversion is relatively cheap in terms of area. The MUX and handshake synchronisers require additional area for the control bits and the associated FSMs. The FIFO controller requires read and write pointer generation and synchronisation plus empty and full flag generation from these pointers, resulting in a large memory and control overhead.

An estimation of register transfer level (RTL) implementation difficulty for each synchroniser is also shown in the table. The fundamental synchroniser is very simple to implement as it is merely a matrix of flip-flops. The Gray-encoded fundamental synchroniser is simply the addition of Gray-to-binary and binary-to-Gray conversion unless the designer implements logic that generates and uses Gray encoded signals directly. The MUX and handshake synchronisers require implementing FSM and request-acknowledge detection logic. The FIFO requires implementing pointer generation logic, auxiliary synchronisers that send the pointers across domains and status flag generation from the pointer values plus connection to and possible implementation of the shared dual-port memory.

2.4 CDC Verification

Clock domain crossing verification provides interesting problems that have been the subject of various studies. There are two main reasons for the interest in this subject.

Firstly, there is the inherent difficulty behind performing CDC verification. Clock domain crossing issues arise from the physical proprieties of integrated circuit latches so they do not manifest during typical digital system simulations, requiring analog simulation instead. Analog simulation of latch metastability is costly in terms of computational resources as it requires

simulation precision levels in the order of femtoseconds (in the current technology node range of 22nm – 7nm). As it is not feasible to perform analog simulation in large circuits, other methods must be explored.

Secondly, there is an opportunity to improve verification efforts by automating some parts of it. Clock domain crossing zones can be identified early and automatically by RTL analysis tools [11] by tracing the clock domain affection range and identifying paths sampled by different clock signals.

There are two main approaches used to perform CDC verification:

- **Conventional testbench with injected CDC jitter:** This approach focuses on simulating the main effect of metastability within the synchroniser RTL logic.

  On testbench environments, the fundamental synchroniser can be modified to randomly insert extraneous delay on domain-crossing bits, effectively simulating CDC jitter [29]. Bad synchronisation modules would be unable to compensate for this jitter and would cause data corruption.

  This approach is advantageous because it is straight forward and familiar to verification engineers. The main issue is that it is highly vulnerable to human error. Since CDC jitter is injected in the synchronisation blocks, completely unsynchronised paths go unchecked by this method. It also does not account for the frequency of metastability events, leaving CDC glitching issues unchecked.

- **Automated checking:** As CDC paths can be identified automatically, it is possible to implement processes that perform automated checks on them. They typically consist of two main types of checks: structural and functional [29].

  Functional checks typically consist in looking at the input and output of the synchronisation modules and checking for data equivalence in both domains. These checks tend to rely on simulation-based solutions that detect data integrity issues located inside the synchronisation module.

  Structural checks analyse the flow of CDC data in and around the synchronisation blocks. It is typically used to detect complete lack of synchronisation, glitching issues at synchroniser inputs and data reconvergence issues on the outputs of multiple synchronisers in one domain.

  Synchroniser design intent can be used by the analysis tool to perform more sophisticated checks. If the tool knows the synchroniser topology that the designer intends to implement (either by requiring the designer to specify them or by automatically inferring the synchroniser type [17, 19]), it can perform structural checks against known valid synchroniser types. Additionally, these checks can be further improved through the use of formal verification methodology to obtain mathematical proof of good synchronisation behaviour [18].

2.5 Summary

This chapter presented an overview of clock domain crossing in the perspective of the issues associated with it in section 2.1 and the standard digital system constructs used to overcome them in section 2.2.

Section 2.3 presents a direct comparison between all mentioned synchronisers. Table 2.1 can be used as a quick reference to allow digital system designers to decide what synchroniser is best for a particular application. Section 2.4 provides a small overview of the usual methods that are employed to verify CDC.

For readers more familiarised with clock domain crossing that do not intend to to explore the chapter in-depth, we would like to highlight a few chapter quirks:

- Fundamental synchroniser formulation

  The simplest synchroniser was presented as a N x M matrix of flip-flops (section 2.2.1). In most studies, this synchroniser is referred to as two-FF or three-FF synchroniser, which translate into 2 x 1 and 3 x 1 fundamental synchronisers, respectively. In this dissertation, we expanded this synchroniser type to a generic matrix in order to further explore the impact of increasing data width (M) and the number of pipeline stages (N).

- FIFO synchronisation notes

  Most studies do not go in-depth into the pointer generation and synchronisation structures in the asynchronous FIFO. During this dissertation, these concepts were explored in some detail as the CDC segmented buffer controller will synchronise in a similar fashion. Section 2.2.4.1 touches on why and how the synchronisation of read and write pointers works to generate full and empty status flags, effectively allowing closed-loop control of the FIFO data by both domains.

Chapter 3

Approach

Chapter Outline

THIS chapter provides a bridge between the theoretical and practical portions of the thesis. With the main dissertation goals outlined in chapter 1 and the CDC background in chapter 2, this chapter presents a formulation of the proposed module architecture.

Firstly, an overview of the architecture and the data to synchronise is presented. Afterwards, the focus of the chapter lies in investigating which synchronisers fit best into the architecture, and thus preliminary implementation and testing of synchronisers will begin here. The goal of the chapter is to allow arrival at a final synchronisation architecture.

3.1 Functional Requirements Overview

Structuring the requirements introduced in chapter 1, figure 3.1 shows the use-case diagram for the interactions in which the segmented buffer controller partakes. The segmented buffer controller is responsible for translating the instructions received from the push and pop sides into RAM read and write instructions and providing packet availability information.

When requesting a packet push or pop, the actor informs the segmented controller of the associated packet ID which the segmented buffer controller then translates into the corresponding memory addresses.

Push and pop use-cases redirect the request to the correct write or read memory address while generating the next access pointer. While the pop side can only pop an element from a

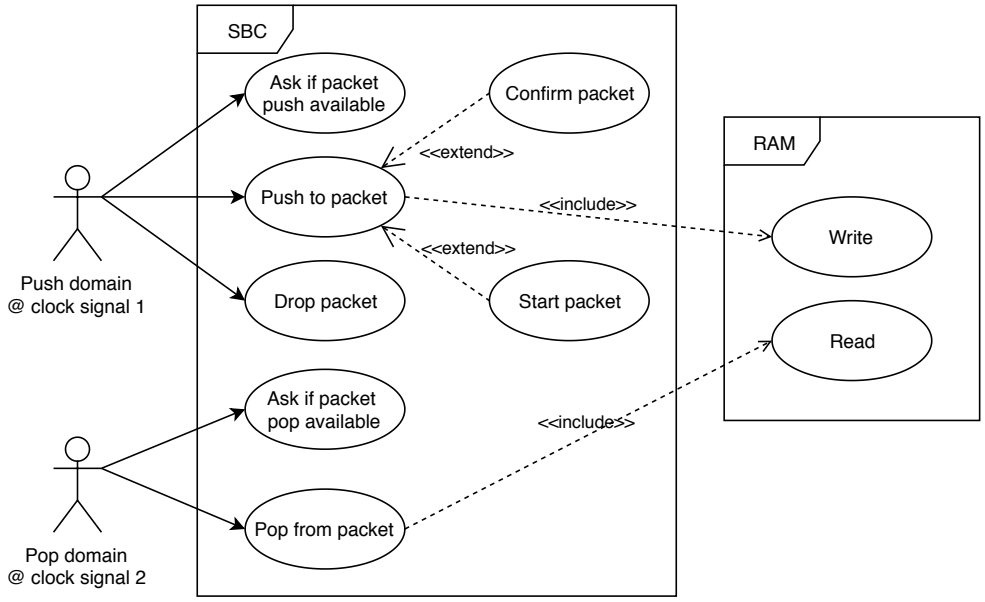


Figure 3.1: SysML use-case diagram of the segmented buffer controller

packet, the push side may choose various request types. It can choose between dropping the current packet, which erases all unconfirmed elements, or push a new element to the packet. When pushing to a packet, the push side may optionally signal that the current push is the start of a new packet, or confirm all elements since the previous start.

In order to ensure safe behaviour, each actor must keep track of packet status through the “ask if packet push available” and “ask if packet pop available” use-cases. The actor can push to a packet if the corresponding segment is not full, and can pop from a packet if the corresponding segment is not empty. For each side to know this packet status, synchronisation is required.

3.2 Proposed Architecture

Figure 3.2 shows the block diagram of the proposed module architecture. The architecture aims to provide resilience against implementation mistakes by clearly separating the design into three types of sub-blocks: push-synchronous blocks, pop-synchronous blocks and synchronisation blocks. Push-synchronous blocks can only use the push clock, pop-synchronous blocks can only use the pop clock, and only the synchronisation blocks may use both. To clarify, “push clock” and “pop clock” refer to the clock signals that drive the push domain (packet transmission) and pop domain (packet reception), respectively.

The architecture allows for the implementation of sub-modules with a variable degree of reliance on synchronised data. Pointer control does not require any synchronised data as it is solely dependent on input stimulus (although the input stimulus should be dependent on the

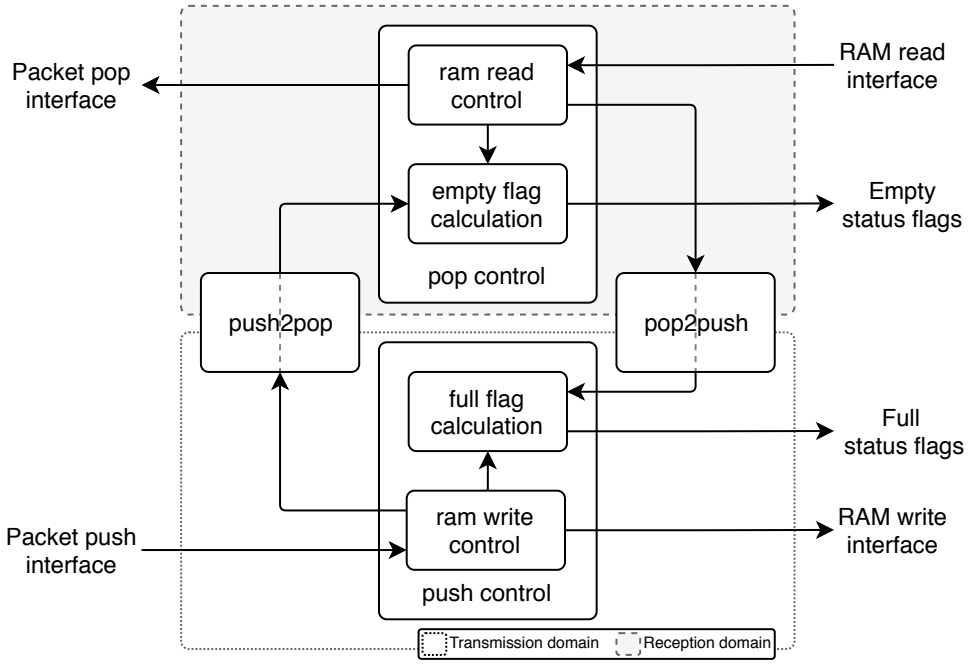


Figure 3.2: Block diagram of the proposed segmented buffer controller architecture

status flags). In turn, status flag calculation is dependent on synchronised data as described in section 2.2.4.1.

The synchronisers will be used to transmit pop activity into the push domain and push activity to the pop domain. Designing these synchronisers is the key issue highlighted in this dissertation and will thus be detailed in sections 3.3, 3.4 and 3.5.

3.2.1 Data to Synchronise

The synchronisation problem is essentially an extension of the synchronisation problem for an asynchronous FIFO where the FIFO push and pop sides synchronise write and read pointers in order to generate full and empty status flags.

Our goal is similar but has some added complexity. Firstly, pointer generation is not linear because the push domain can jump multiple pointer positions in the same clock cycle due to packet confirmation or cancellation. Secondly, since the real read and write pointers are associated with a single physical RAM, but status flag calculation is associated with a virtual FIFO that does not have the same read and write addresses, the read and write pointers cannot be synchronised directly.

The data to synchronise for status flag generation is the virtual FIFO read and write pointers concatenated with the wrap-status bit as described in section 2.2.4.1. In order to distinguish these from the real RAM pointers, they will henceforth be referred to as read and write counters. The array of read counters generated by the pop domain should be synchronised into

the push domain, and the array of write counters generated by the push domain should be synchronised into the pop domain.

Figure 3.3 shows the structure of the array of virtual FIFO counters and its correspondence to the physical RAM. Each packet is stored in one memory segment and to it corresponds one read counter and one write counter. To each counter value corresponds one physical memory address. By accessing these counter values, each side can know the segment status (full or empty).

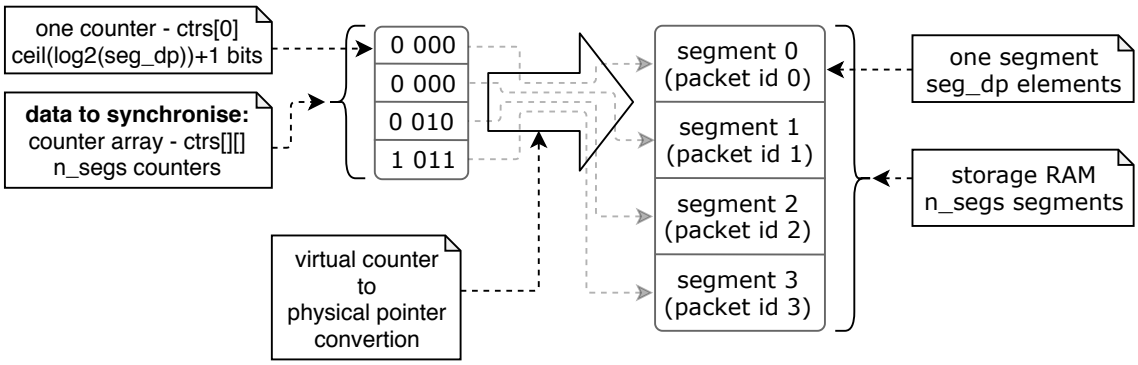


Figure 3.3: Synchronised data correspondence to RAM (4 packets with 8-element MTU)

In order to provide explicit references to the counter values at each domain, we will use the terms “local” and “remote” to refer to the counters in each domain. For example, the local write counter is the write counter value on the push domain, and the remote write counter is the write counter value on the pop domain which will be delayed in relation to the local write counter. Furthermore, under the context of the synchronisation block which can be implemented in any direction (from push to pop or from pop to push), “reception domain” refers to the domain that is receiving synchronised data (not to be confused with the packet reception domain), and the same logic applies for mentions of the “transmission domain”.

3.3 Synchroniser Candidates

The data to synchronise is simply an array of counters, where we highlight two main approaches: integral synchronisation where the entire counter array is synchronised as a whole, and partial synchronisation where only one virtual FIFO counter is synchronised at a time.

3.3.1 Gray Synchroniser

The most common synchronisation scheme for pointer synchronisation is the Gray coded synchroniser. Figure 3.4 shows how the synchroniser can be applied to this synchronisation context. The advantage of this synchroniser is that, when used correctly, it is not vulnerable to data incoherency while providing fast synchronisation due to its open-loop nature.

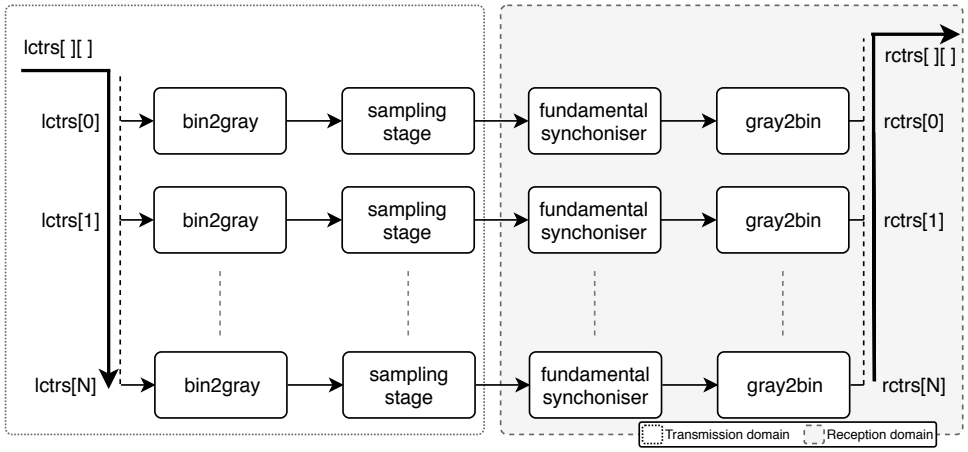


Figure 3.4: Block diagram of the Gray synchroniser for the proposed architecture

Limitations and Restrictions

The main limitation of this mechanism is that the local counter may only increment or decrement by at most 1 in each clock of the transmitting clock domain.

In the receiving clock domain, the signal does not have this limitation (that is, it can increment or decrement any arbitrary number of positions in the perception of the reception clock domain). This is because, at any one particular reception clock edge, there can only be at most 1 bit vulnerable to metastability (metastability only occurs when both clock edges match, and only one bit changes every transmission clock edge). If the bit does go metastable, we either sample the previous or the next counter, both being valid.

This limitation is typically not a problem for asynchronous FIFOs as the pointers do tend to follow these rules. However, some FIFO implementations require managing arbitrary pointer jumps. Some examples:

- Resetting. For asynchronous resets this is typically not a big issue, however for synchronous resets (i.e., functional re-initialisation requests) this would require dedicated handshake mechanisms to ensure safe a reset.

- Complex FIFO functionality. Allowing a side to move multiple elements at once means jumping multiple positions in its pointer. This can, for example, be eliminated by dampening the pointer transitions on the transmitting domain at the cost of added latency.

- Non-power-of-two FIFO depths. For Gray encoding to synchronise correctly, pointer generation must be implemented such that only one bit is toggled on wrap-around. As shown in figure 2.5, this is possible for segment depths that are multiples of two. Otherwise, we must start counters with a static offset, which also has implications for RAM address and status flag calculation.

Area Considerations

Area cost depends on the counter and pointer generation logic. All counter and pointer generation and utilisation logic can be implemented in Gray code, eliminating the need for Gray code conversion (gray2bin, bin2gray). However, arbitrary status flag calculation (e.g., 'full minus one' and 'empty plus two') can be challenging in this codification. Due to this, and other flexibility issues, we will not explore this possibility and instead opt for the implementation of Gray encoding and decoding within the synchroniser.

The area cost is simply the cost of the fundamental synchronisers plus an additional sampling stage before the synchronisers and, if Gray encoding is implemented, the cost of the encoding and decoding logic. The additional sampling stage is required in order to prevent glitches in the CDC path as described in section 2.1.1.

Performance Considerations

This synchroniser has very low performance overhead which consists of the latency of the sampling stage plus the latency of the fundamental synchronisers.

If pointer dampening is implemented as previously described, there is an added latency of one transmission domain clock cycle per jumped word.

3.3.2 Handshake Synchroniser

The read and write counters can be synchronised without codification through a handshake mechanism. Figure 3.5 shows the handshake synchronisation scheme for this context. Like the Gray synchroniser, this synchroniser performs integral data synchronisation.

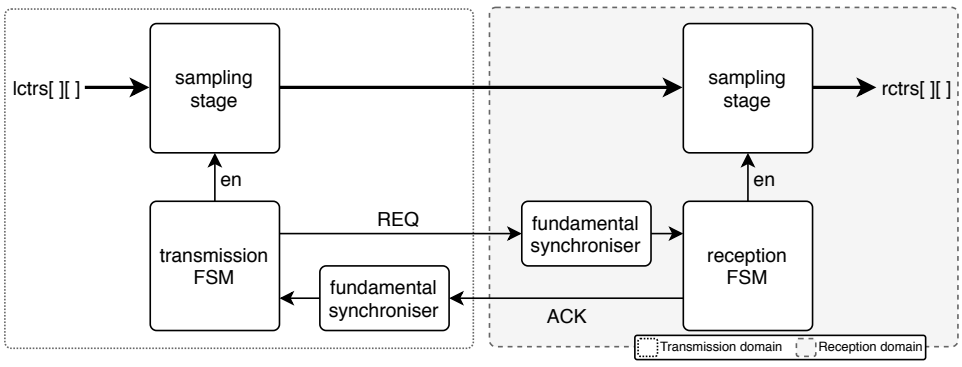


Figure 3.5: Block diagram of the handshake synchroniser for the proposed architecture

In this configuration, a state machine is used by the transmitting side to load the transmission domain data sample register. This state machine can have only two states (waiting state and loading state).

The loading state is active for one clock cycle and loads the transmission domain data into the sample register. One clock cycle later, the state machine enters the waiting state and the

handshake request signal is asserted. When the reception domain receives the request, its data sampling stage is enabled for one clock cycle, and an acknowledge signal is generated. When the transmission domain receives the acknowledgement, its state machine moves again to the loading state and the algorithm repeats.

Limitations and Restrictions

Unlike the Gray coded synchroniser, this design does not have any significant restrictions. The tradeoff for this flexibility is reflected in the additional area and performance overhead.

Area Considerations

The reception data sample register output can be used directly by the reception clock domain as it is not vulnerable to metastability. The transmission data sample register must be added to hold a copy of the pointer values stable during synchronisation. The transmission state machine and request/acknowledge generation and detection do not have significant area overhead.

Performance Considerations

The handshake mechanism causes a significant performance impact. Between each transmission, there are at least two reception clock cycles until the request is detected and two transmission clock cycles until the acknowledgement is detected.

3.3.3 Onehot Synchroniser

A method to avoid the restrictions behind Gray coding while maintaining synchronisation through an array of fundamental synchronisers is to one-hot encode the addresses instead, as shown in figure 3.6. Like the Gray and handshake synchronisers, this synchroniser performs integral synchronisation.

Since between any two reception domain data changes there are always two one-hot bit toggles, the reception domain can trivially detect whether there has been metastability data corruption by counting the amount of set (equal to 1) bits in the signal.

Figure 3.7 illustrates the possible transitions between two one-hot-encoded states. The reception domain can perform data validity checks by calculating the exclusive-or between all bits of the received signal. The only case where data is lost is when all sampled bits are reset (equal to 0). If two bits are set, the reception data is correctable by resetting the old bit.

Limitations and Restrictions

This synchronisation scheme does not present any notable functional restrictions. However, this synchronisation method can be prohibitive in terms of area, and is only be feasible for small virtual FIFO sizes.

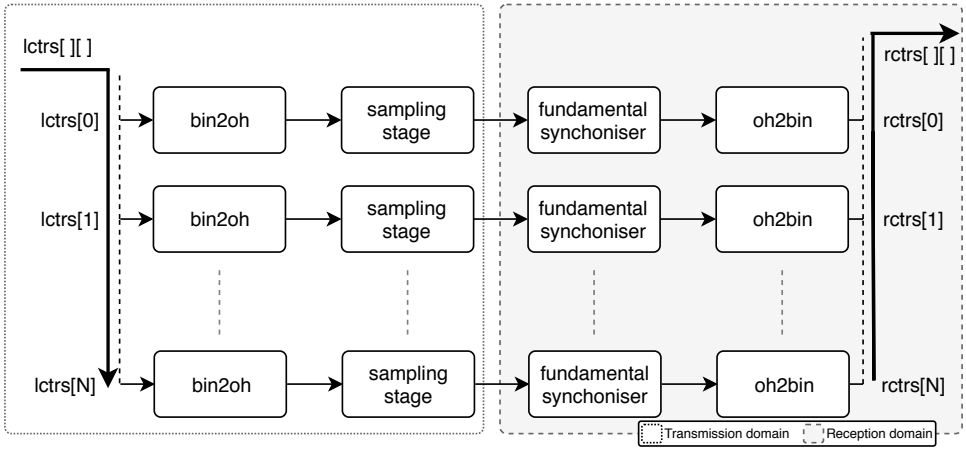


Figure 3.6: Block diagram of the one-hot synchroniser for the proposed architecture

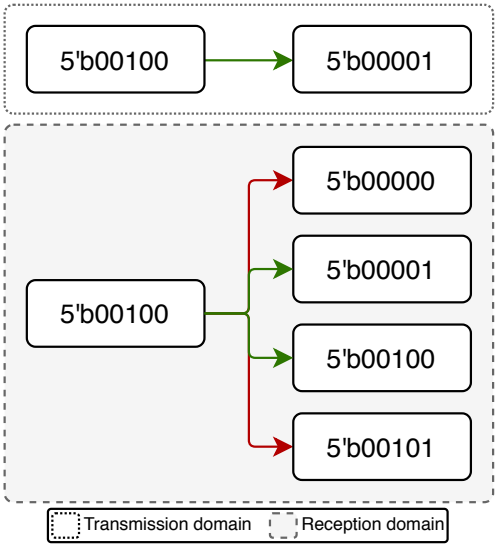


Figure 3.7: Possible onehot-encoded CDC transitions in each re-sample

Area Considerations

Firstly, the signals must be encoded from binary to one-hot in the transmission domain and encoded back to binary in the reception domain. It is possible to calculate status flags using one-hot encoded signals, removing the need to decode the signal, but the reception domain would require logic to compare binary encoded counters with one-hot encoded counters.

The one-hot encoded counters have a size of twice the virtual FIFO depth (in number of bits) which must be sampled after encoding to prevent glitching in the CDC path. This number of bits is required as the concatenation of the wrap-bit with the binary counter results in a duplication of bits in one-hot codification. Then, there is the added cost of running the signals through an array of fundamental synchronisers. While the other synchronisers scale logarithmically with the segment size, this synchroniser scales linearly.

Performance Considerations

The performance impact of this synchroniser is low, although slightly higher than the Gray-coded synchroniser as it is still vulnerable to metastability data loss. When metastable data corruption occurs the receiving clock domain discards the data. When no metastability occurs, there is no performance difference when comparing to the Gray-coded synchroniser.

3.3.4 Shared Synchronisation FIFO

A single synchronisation FIFO may be shared across all virtual FIFOs in other to transmit counter update information. The word stored in the synchronisation FIFO would pack the data referring to which virtual FIFO counter to update and the desired corresponding value. This word stores counter update information relative to one counter value update and allows the implementation of partial synchronisation, as shown in figure 3.8.

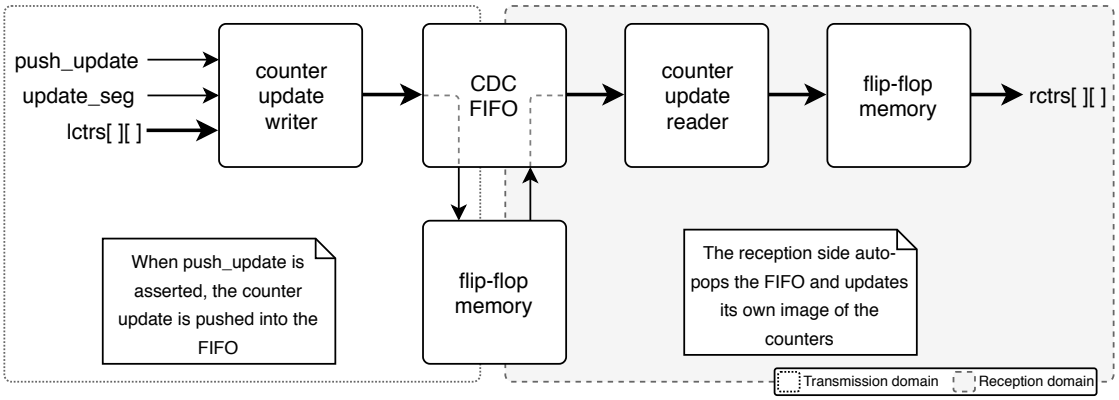


Figure 3.8: Block diagram of the FIFO synchroniser for the proposed architecture

Limitations and Restrictions

The shared synchronisation FIFO does not have any important functional limitations. However, it does have the most difficult implementation out of all proposed synchronisers.

It benefits from some additional signals from the transmission domain. In particular, a bit that indicates that a counter update was performed and a bus that indicates in what virtual FIFO this counter update happened. Although this information can be obtained just by monitoring the counter array, it would needlessly increase area and make high-frequency synthesis more difficult. Secondly, the transmission domain must be able to accept a halt signal from the synchroniser that causes all transmission domain activity to stop.

Area Considerations

Most of the area cost comes from the synchronisation FIFO memory. The depth of memory is configurable and equal to the maximum pending updates that are allowed to reside in the synchronisation FIFO. Once the synchronisation FIFO is full, the synchroniser must halt the transmitting domain because pointer update information can no longer be stored.

If trying to avoid any transmission side halting, the worst-case scenario is when the transmitting domain fills the entire FIFO one-by-one before the receiving domain pops any data, which means the FIFO depth has to match the total word count stored in the segmented buffer RAM, which results in a prohibitive area overhead. However, if we know that the receiving domain is working faster than the transmitting domain, the FIFO only needs to hold enough words to compensate for the synchronisation latency, resulting in a very manageable area cost.

Performance Considerations

The FIFO synchroniser does not incur in a performance hit unless the FIFO memory fills, in which case it causes halting of the transmitting domain. This architecture is effectively modelled by leaky-bucket congestion, which will be elaborated in section 3.5.1.

3.4 Fundamental Synchroniser MTBF

Before performing synchroniser tests, the number of fundamental synchroniser stages must be chosen according to a desired MTBF, as this choice impacts both the performance and the area of the synchroniser.

Please note that this section only aims to provide a rough estimate for the MTBF values as obtaining accurate values would require going in-depth into analog latch physics and average bit toggle frequency statistical analysis which falls outside of the intended scope of this dissertation.

A target MTBF of 10000 years was chosen, and equation 2.5 will be used to get the target number of pipeline stages [8, 20]. In order to solve the equation, some implementation details still need to be exposed.

Firstly, we must define the asynchronous data frequency (f_d) and the sampling clock frequency (f_c). Through equation 2.2 we can conclude that MTBF decreases with an increase in either frequency. Therefore, the most pessimistic MTBF scenario is when both frequencies are at their highest possible values. Since this module is designed for compatibility with PCIe, the latest PCIe specification was used to determine maximum frequencies, which translates into 1 GHz for both domains [22] (equation 3.1).

Following this, t_r is defined as the maximum allowed resolution time for each latch in the sequence of flip-flops. It is equal to half of a sampling clock cycle (equation 3.2).

$$f_d = f_c = 1 \cdot 10^9 \quad [\text{Hz}] \tag{3.1}$$

$$t_r = \frac{1}{2 \cdot f_c} = 5 \cdot 10^{-10} \quad [\text{s}] \tag{3.2}$$

Lastly, the technological variables τ and T_0 must be defined. These values typically depend on analog circuit simulations with very high precision sweeps between clock and data edges.

Table 3.1 shows the resulting MTBF for some target technologies in function of the number of fundamental synchronisation pipeline stages. The table also shows the minimum frequency at which it is necessary to increase the number of pipeline stages to 3. This table is based on Synopsys simulations for the smallest latches with $V_{supply} \in [0.625, 0.670]\ V,\ T = -40C$. MTBF increases with the increase in supply voltage, however, obtaining accurate values for higher voltages usually becomes too computationally intensive as the metastability window tightens, therefore these are pessimistic calculations in terms of the supply voltage.

Table 3.1: Fundamental synchroniser MTBF per technology and synchroniser stage depth

Technology	MTBF (1 FF)	MTBF (2 FF)	MTBF (3 FF)	min_f_{3FF}
TSMC 22nm FinFET LVT	$709\ yrs$	$4.76 \cdot 10^{20}\ yrs$	$3.19 \cdot 10^{38}\ yrs$	2.25 GHz
TSMC 16nm FinFET LVT	$2590\ yrs$	$5.71 \cdot 10^{23}\ yrs$	$1.26 \cdot 10^{43}\ yrs$	2.44 GHz
TSMC 7nm FinFET LVT	$6.7 \cdot 10^{6}\ yrs$	$1.56 \cdot 10^{28}\ yrs$	$3.64 \cdot 10^{49}\ yrs$	2.46 GHz
TSMC 7nm FinFET ULVT	$3.66 \cdot 10^{18}\ yrs$	$2.95 \cdot 10^{51}\ yrs$	$2.38 \cdot 10^{84}\ yrs$	3.94 GHz

The target technology for this module is 16 nm. Even for 22nm, both f_c and f_d would have to rise to approximately 2.25 GHz for a three flip-flop synchroniser to be required to meet the MTBF requirement.

The aggregated MTBF of the entire synchronisation stage can be calculated through equation 2.6. According to this equation and for TSMC 22nm FinFET we would need approximately 10^{15} bits for more synchroniser stages to be required. Therefore, for this application and technology nodes, two-stage fundamental synchronisers were chosen.

3.5 Synchroniser Evaluation

From the synchroniser proposals mentioned above, we already have an idea of which synchronisers will perform better depending on circumstances. These tradeoffs will be analysed in further depth in order to understand their application consequences more clearly.

Three test scenarios were picked in order to obtain points of comparison from experimental tests, as shown in table 3.2. These test scenario parameters define the segmented buffer dimensions and are extracted from real PCIe device configurations with different levels of data rate and complexity. Although in scenario A both frequencies are 500 MHz, the clocks are expected to have different phase, requiring synchronisation.

Table 3.2: Test scenarios used for synchroniser evaluation

Scenario	Seg. Depth	# Segs	Data Wd.	Fast freq.	Slow freq.
A	18	4	67	500 MHz	500 MHz
B	18	32	133	500 MHz	125 MHz
C	18	256	265	1000 MHz	62.5 MHz

3.5.1 Area Evaluations

Shared Synchronisation FIFO Depth

The FIFO synchroniser is the only synchroniser that is flexible in terms of allowing parameterisation that directly correlates to its area. The depth of the shared synchronisation FIFO can be modelled by the "leaky bucket as a queue" transmission topology [1, 2], as seen in figure 3.9.

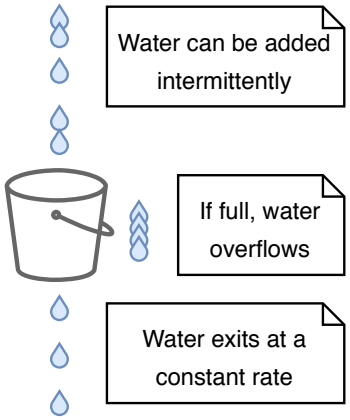


Figure 3.9: Concept of leaky bucket, analogous to data transmission concepts

In this analogy, water (data) arrives at an unknown rate, the bucket (synchronisation FIFO memory) holds the maximum burst that is allowed, and water (data) leaves at a constant rate.

The synchronisation FIFO depth is equivalent to the size of the bucket. If the bucket fills, additional water will leak out (if the memory fills the data will be lost).

The only way to guarantee that the shared FIFO memory does not fill is to never allow the data rate at the transmitter to exceed the receiver data rate capacity (equation 3.3). In this equation, a_{tsmt} is the transmitter activity, i.e. the average amount of clock cycles it is actively pushing/popping divided by the total amount of clock cycles (equation 3.4).

$$a_{tmst} \cdot \frac{f_{tsmt}}{f_{recv}} < 1 \tag{3.3}$$

$$a_{tsmt} = \lim_{N \to \infty} \left(\frac{N_{active}}{N_{active} + N_{idle}} \right) \tag{3.4}$$

Equation 3.3 is always fulfilled when $f_{tsmt} < f_{recv}$. In this case, the FIFO depth is the maximum latency between domains through the fundamental pointer synchronisers, as the FIFO will never fill beyond this latency. If this condition is not met, the FIFO depth dictates the maximum burst at the transmitting domain.

Area tests were performed for two FIFO depth configurations: FIFO 1 (8 depth) and FIFO 2 (depth equal to segment depth). FIFO 1 is sized for slow-to-fast synchronisation while FIFO 2 allows the transmitting domain to burst an entire segment before synchronising to the other domain. FIFO 1 depth was obtained through a simulation where both the transmission and reception domains are working at maximum activity and, sweeping across frequencies where $f_{tsmt} < f_{recv}$, the maximum observed number of pending counter updates was recorded.

Clock Domain Crossing Data Width

Although all synchronisers result in the transmission of the same data, the internal data width that crosses domains and is vulnerable to metastability is different across the synchronisers. The area scaling of the synchronisers across domain boundaries is related to the scaling of this data width.

For small segmented buffers, CDC data width will not be significantly different across the synchronisers. However, as the segmented buffer size increases, area differences caused due to a discrepancy in domain crossing data width become noticeable.

For the Gray-code and handshake synchronisers, the synchronisation data width is the same as the input data width (equation 3.5).

For the one-hot synchroniser, the data is one-hot-encoded and requires more area in the CDC boundary, according to equation 3.6.

The FIFO synchroniser only crosses the segment identifier and current counter for that segment, resulting in equation 3.7. This equation does not account for the shared synchronisation FIFO pointer synchronisation area, which is a constant and dependant on the FIFO depth parameterisation.

$$data\_wd_{gc,hs} = n\_segs \cdot (\lceil \log_2(seg\_dp) \rceil + 1) \quad \text{[bit]} \quad (3.5)$$

$$data\_wd_{oh} = n\_segs \cdot (2\ seg\_dp) \quad \text{[bit]} \quad (3.6)$$

$$data\_wd_{fifo} = \lceil \log_2(n\_segs) \rceil + (\lceil \log_2(seg\_dp) \rceil + 1) \quad \text{[bit]} \quad (3.7)$$

The CDC data widths for each scenario are listed in table 3.3. The FIFO synchroniser has the best data width results, which is expected as the data width scales logarithmically with both the number of segments (n_segs) and the segment size (seg_dp). The Gray and handshake synchronisers scale linearly with the number of segments instead. The One-hot synchroniser scales linearly with both parameters.

Table 3.3: CDC data widths observed per test scenario

Synchroniser	$data\_wd_{4x18}$	$data\_wd_{32x18}$	$data\_wd_{256x18}$
FIFO	12	30	48
Gray	24	192	1536
Handshake	24	192	1536
Onehot	144	1152	9216

The data width efficiency of the FIFO and One-hot synchronisers, relative to the data width of the Gray and Handshake synchronisers, is resolved into the equations 3.8, 3.9 and plotted in figure 3.10.

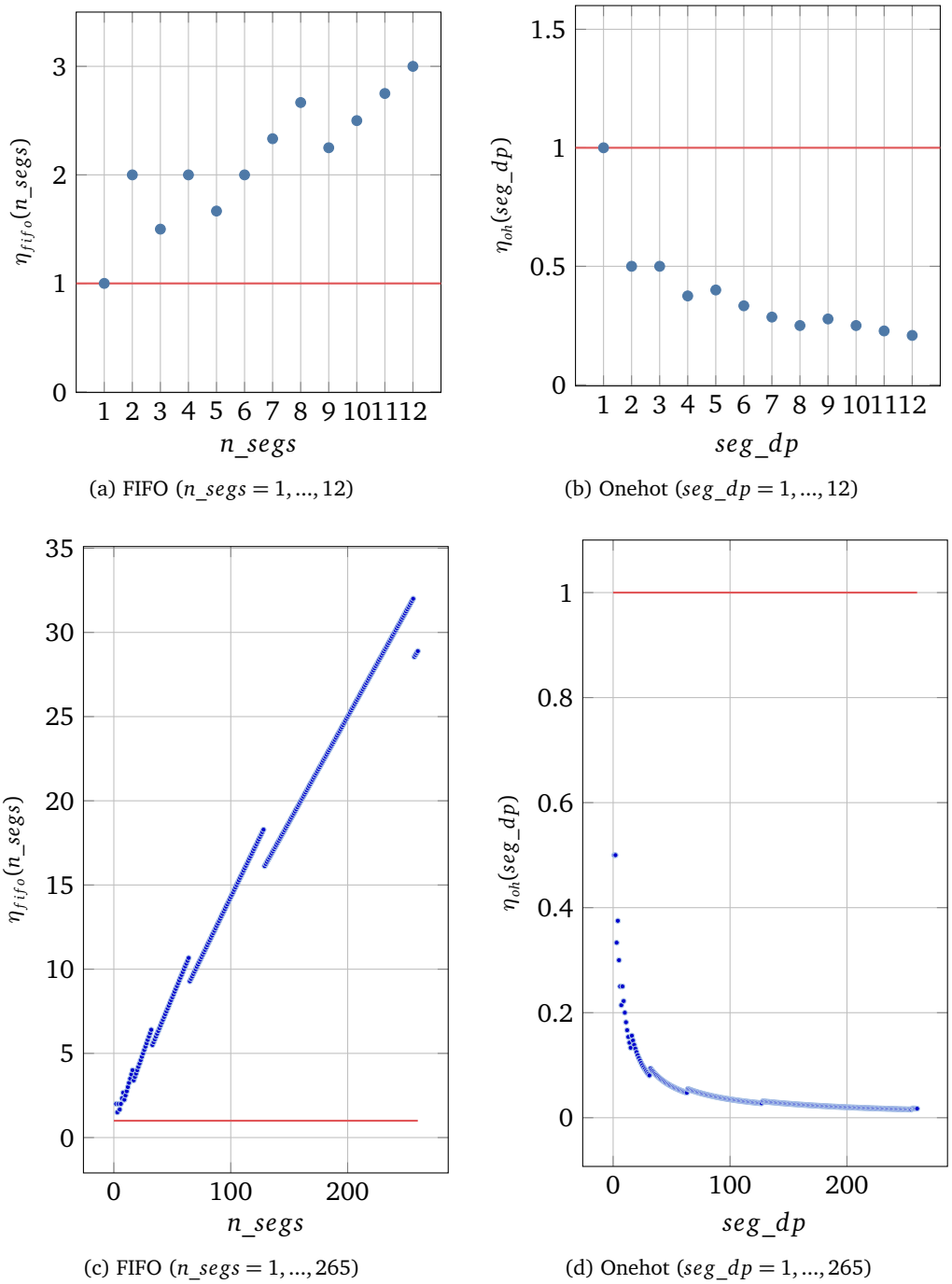
$$\eta_{fifo}(n\_segs) = 100\frac{n\_segs}{\lceil \log_2(n\_segs) \rceil},\ \forall\ n\_segs\ \in \mathbb{N} > 1 \quad [\%] \quad (3.8)$$

$$\eta_{oh}(seg\_dp) = 100\frac{\lceil \log_2(seg\_dp) \rceil}{2\ seg\_dp},\ \forall\ seg\_dp \in \mathbb{N} > 1 \quad [\%] \quad (3.9)$$

Note that equations 3.8 and 3.9 are not defined for $n\_segs = 1$ and $seg\_dp = 1$ respectively, although figure 3.10 plots the functions at these values. This is because $log2(1) = 0$ but the value was considered to be equal to 1 for these plots as it is allows unified RTL across all parameterisation values, although a segmented buffer with either of these variables set to 1 is just a normal continuous memory.

From these plots we can visualise that the onehot synchroniser is only viable for very small segment depths and the FIFO synchroniser becomes increasingly better as the number of segments increase.

Despite being an interesting area reference, CDC data width only accounts for a small part of the segmented buffer controller area. The biggest example of this is the FIFO synchroniser that requires its own dedicated memory and does not follow the typical fundamental synchroniser matrix configuration seen in the other synchronisers. Although we now know what



(a) FIFO ($n\_segs = 1, ..., 12$)

(b) Onehot ($seg\_dp = 1, ..., 12$)

(c) FIFO ($n\_segs = 1, ..., 265$)

(d) Onehot ($seg\_dp = 1, ..., 265$)

Figure 3.10: Synchroniser tests – Relative synchroniser CDC data width

results to expect, the area cost of each synchroniser will be further explored through synthesis area results.

Post-Synthesis Area

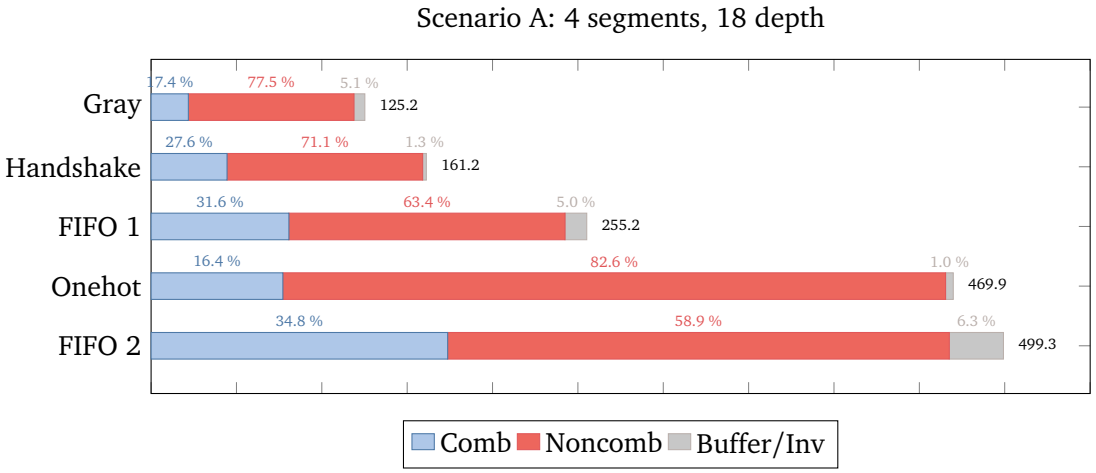
The synchroniser candidates were implemented and synthesised in order to evaluate the full area cost of each synchroniser. Synthesis was performed in Design Compiler [10] with Ultra High Effort and both clock paths constrained to 1 GHz at 16nm TSMC FinFET technology. The synthesis report data was grouped per scenario and plotted in figure 3.11.

In general, the synthesis results correspond to the expected. In scenario A, the Gray synchroniser has the lowest area cost as the only overhead is the combinational conversion to and from Gray code.

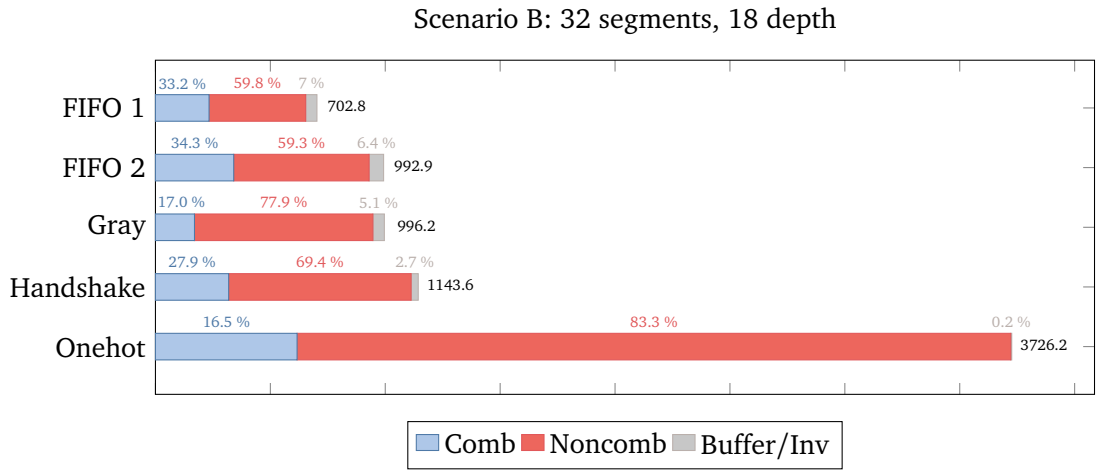
The FIFO synchroniser shows poor results in scenario A due to the required dedicated memory overhead that stores pending updates. As the segmented buffer size increases, the FIFO synchroniser becomes progressively better, and in scenario C it is the best synchronisers in terms of area. This is mostly due to its CDC data width, which is the smallest out of all synchronisers.

The One-hot-coded synchroniser has poor results starting in scenario A that get progressively worse as the number of segments increases. Due to this area cost, the synchroniser was considered not to be viable for further implementation.

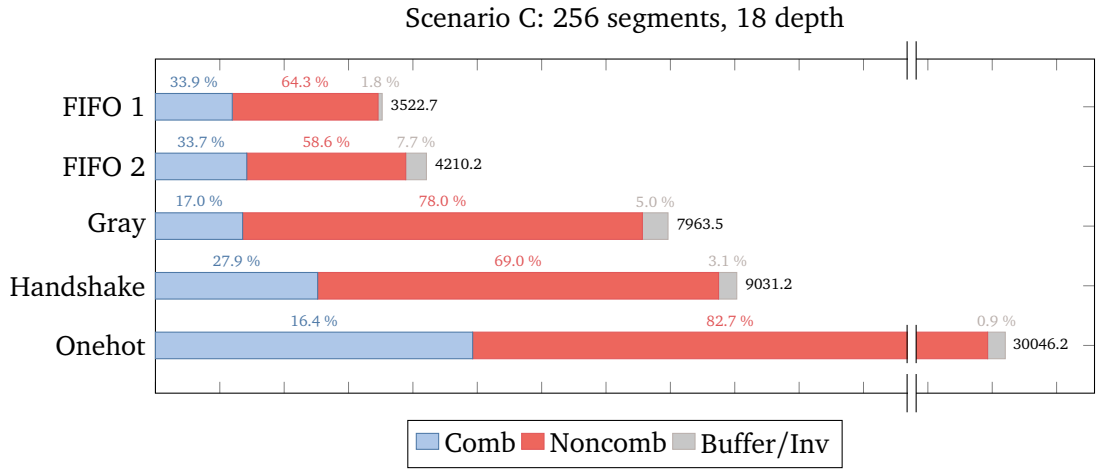
The handshake synchroniser presents average area results, typically placing it in the middle of the ranking and slightly below the Gray synchroniser for all scenarios. The handshake synchroniser provides good results both in functional restrictions and limitations and in area. However, its biggest drawback is the lower synchronisation speed as it is the only synchroniser where the data path is halted by closed-loop control signal synchronisation. A comparison of synchroniser performance will be presented in section 3.5.2.



(a) Scenario A



(b) Scenario B



(c) Scenario C

Figure 3.11: Synchroniser tests – Post-synthesis area results (in FFs)

3.5.2 Performance Evaluations

Synchronisation Speed

Figure 3.12 shows the results of a synchronisation speed performed through through functional simulation. A testbench was developed to count the number of successful synchronisations in a $1\mu s$ time span, which corresponds to 1000 clock cycles at the maximum 1 GHz clock frequency. A successful synchronisation is considered to be the union of both domains refreshing their synchronisation data. The count of successful synchronisations is the number of acknowledges, n_acks.

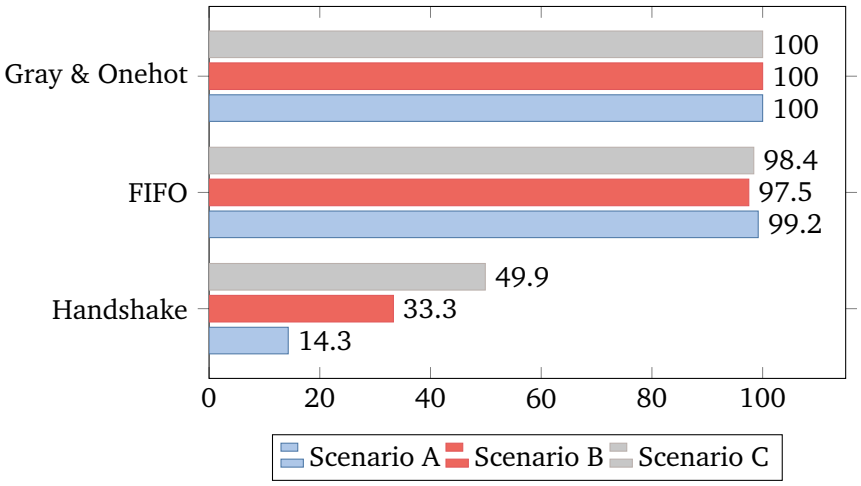


Figure 3.12: Synchroniser tests – Relative (%) synchronisation speed simulation results

Only steady-state speed is measured, which means that for the non-handshake synchronisers, it is assumed that synchronisation data is already arriving at the start of the simulation.

For the handshake synchroniser, the acknowledge count can be measured directly by incrementing the counter each time the acknowledge signal arrives at the transmission domain. As for the Gray-coded and one-hot-coded synchronisers which are open-loop configurations and always in transmission, n_acks is determined by the number of slow clock cycles (equation 3.10).

For the FIFO synchroniser, the number of ACKs is calculated by counting the amount of reception domain pops (equation 3.11).

$$n\_acks_{gc,oh} = n\_slow\_clock\_cycles \tag{3.10}$$

$$n\_acks_{fifo} = n\_pops \tag{3.11}$$

Figure 3.12 shows the measured synchronisation count for all scenarios as the percentage of the maximum observed acknowledge count for that scenario. The results do not show any significant performance differences between the Gray, onehot and FIFO synchronisers.

The FIFO synchroniser has a slightly lower acknowledge count due to the CDC pessimism present in both the transmission and reception sides where pointer synchronisation latency halts some data refreshments, however, this does not result in a significant performance loss. One characteristic that is not accounted for is that when the FIFO is full, the transmission side is blocked and cannot change its pointers, while the other synchronisers allow the transmission domain to continue working. Furthermore, the higher the word count inside the FIFO, the higher the discrepancy between the transmission and reception domains. These problems do not arise when the transmission domain is slower than the reception domain.

The relative speed of the handshake synchroniser has changed significantly across the scenarios. From these three data points, it seems that the synchroniser becomes better as the discrepancy between frequencies increases. This relative speed efficiency will be further explored.

Handshake Synchronisation Speed Efficiency

The handshake synchronisation control path is characterised as a sequence of events, as shown in table 3.4. These events were also characterised in terms of timing in order to provide a better understanding of how much time is required between two consecutive synchronisations. In this event sequence, event 1 can happen in parallel with the event 9 of the previous synchronisation.

Table 3.4: Control path events in a handshake synchroniser

	Event	Δt	Domain
1	Send intent asserted	–	Transmission
2	Send data sampled	T_{tsmt}	Transmission
3	Send pulse asserted	T_{tsmt}	Transmission
4	REQ outbound	T_{tsmt}	Transmission
5	REQ inbound (meta)	$\phi(t_s \rightarrow r_c, t)$	Reception
6	REQ caught	T_{recv}	Reception
7	ACK outbound	ΔT_{ack}	Reception
8	ACK inbound (meta)	$\phi(r_c \rightarrow t_s, t)$	Transmission
9	ACK caught	T_{tsmt}	Transmission

In table 3.4, T_{tsmt} and T_{recv} are the clock periods of the transmission and reception domains, respectively. Events 5 and 8 span a time of $\phi(t_s \rightarrow r_c, t)$ which is the clock skew from transmission to reception at t, and $\phi(r_c \rightarrow t_s, t)$ which is the clock skew from reception to transmission at t, respectively. This skew is not deterministic and is limited according to equations 3.12 and 3.13.

$$0 < \phi(t_s \rightarrow r_c, t) < T_{recv} \tag{3.12}$$

$$0 < \phi(r_c \rightarrow t_s, t) < T_{tsmt} \tag{3.13}$$

The time ΔT_{ack} is dependent on synchroniser design, in particular the functional timing of the acknowledge response generation. The acknowledge response can be generated in the same clock cycle as the detection of the request signal, which is known as early acknowledge. The problem with early acknowledge is that the acknowledge is already in transit before the data is sampled. If the transmission domain is much faster than the reception domain, then there is a possibility that the transmission domain will trigger a data change before the reception domain has actually finished sampling the data. The safe approach to acknowledge generation is to wait for one clock cycle in order to ensure the data is sampled safely. Therefore, ΔT_{ack} is defined according to equation 3.14.

Knowing this, the delay between two back-to-back transmissions, T_{tr}, is bound according to equation 3.15.

$$\Delta T_{ack} = 0 \ \vee \ \Delta T_{ack} = T_{recv} \tag{3.14}$$

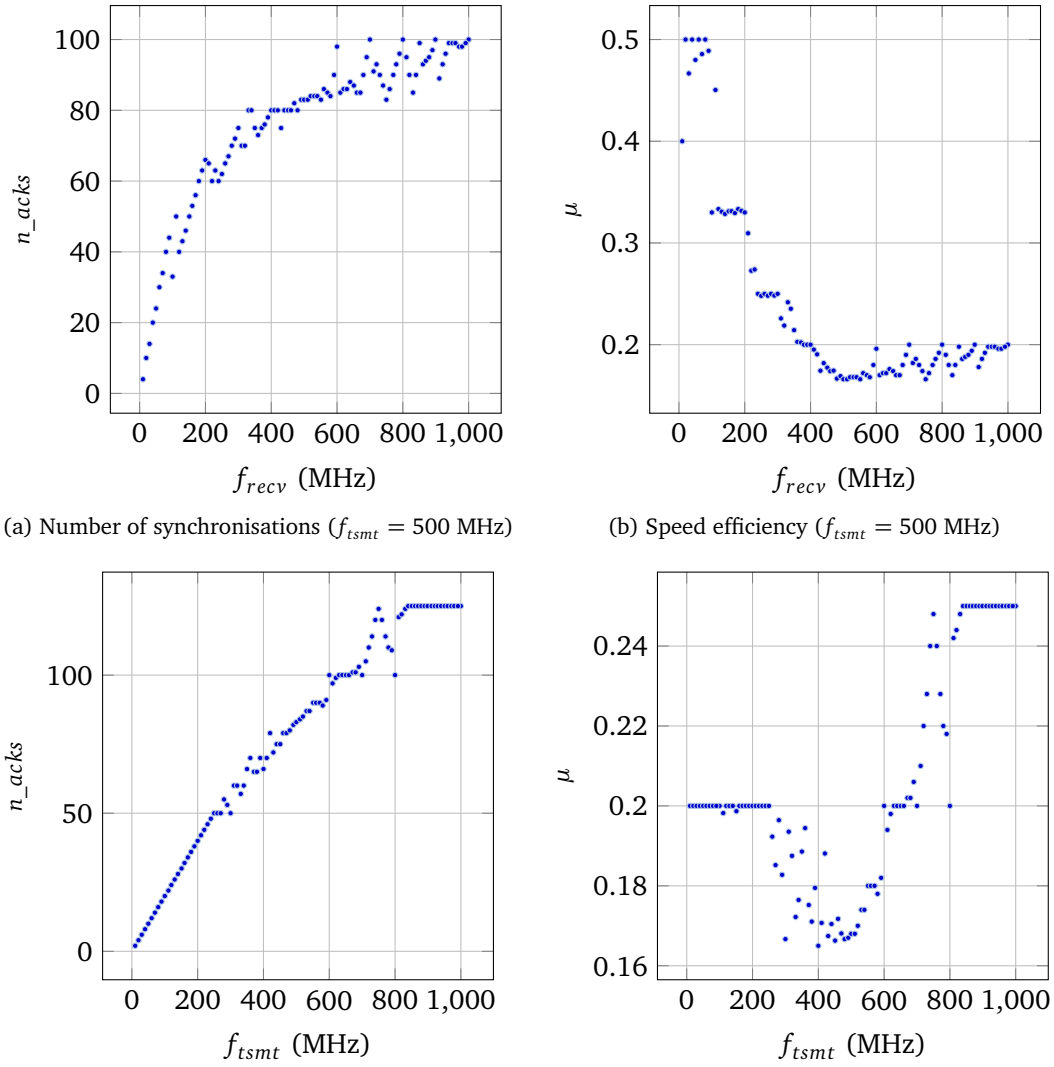
$$4T_{tsmt} + 1T_{recv} + \Delta T_{ack} < T_{tr} < 5T_{tsmt} + 2T_{recv} + \Delta T_{ack} \tag{3.15}$$

While we now know the transmission delay limits between two transmissions, the average transmission delay for any two frequencies is difficult to find analytically. It depends on the average of the instant clock skew values at events 5 and 8, which in turn also depends on initial clock skew.

To obtain the average handshake transmission speed, functional frequency sweep simulations were performed. The testbench initially described in this section was used to measure the number of synchronisations in a $1\mu s$ timespan. The number of synchronisations were measured and compared to the number of synchronisations that the Gray and one-hot synchronisers would perform in the same timespan. The results are shown in figure 3.13.

By analysing the results, a few previous observations can be reaffirmed. Firstly, that the transmission domain has a more considerable impact on the synchronisation speed as can be seen by the speed efficiency plots where, for example, if the transmission domain is much faster than the reception domain then the efficiency is around 50% (figure 3.13b) and if the reception domain is much faster then the efficiency is around 20% (figure 3.13d).

Another interesting remark is that speed efficiency is lowest when the frequencies are similar. This is mostly related to higher average ϕ for these frequency ratios in events 5 and 8 of table 3.4.



(a) Number of synchronisations ($f_{tsmt} = 500$ MHz)

(b) Speed efficiency ($f_{tsmt} = 500$ MHz)

(c) Number of synchronisations ($f_{recv} = 500$ MHz)

(d) Speed efficiency ($f_{recv} = 500$ MHz)

Figure 3.13: Synchroniser tests – Handshake speed frequency sweep simulation results

3.6 Summary

This chapter focused on providing a bridge between theory and implementation. Firstly, section 3.1 described the functionality that the segmented buffer controller module must implement.

An architecture that provides base support for the desired functionality was presented in 3.2. The remainder of the chapter was focused on studying synchronisation schemes that can fit into the architecture, detailing their implementation and trade-offs in sections 3.3 and 3.4. Finally, the synchronisers were further studied in section 3.5 by performing speed and area tests.

With this chapter, we are now on course for full implementation of the segmented buffer controller module with an informed decision on what synchronisers to pick. Presenting the final synchronisation architecture and results from the synchroniser integration will be the goal for chapter 4.

Chapter 4

Implementation and Results

Chapter Outline

FROM the synchroniser candidates and results presented in chapter 3, it is now possible to integrate synchronisers into our architecture (section 3.2) with a clearer understanding of which are better for each situation. The synchroniser choice is dependant on design constraints, which will be quickly re-summarised.

Section 3.1 mentions the primary design constraints. In particular, packets must be confirmed before they are actually transmitted. This means that the write pointers may jump multiple positions in each clock cycle, disallowing the use of the Gray synchroniser. The read pointers do not follow this restriction.

The existence of different constraints for the push and pop sides suggests that a hybrid synchronisation architecture may present the most interesting results. Additionally, the FIFO synchroniser can be useful as it provides the best area results for large segmented buffers.

The implemented module was designed to be parameterisable concerning its CDC architecture according to table 4.1. These architectures aim to provide the best synchronisers for each specific configuration case. The default architecture can be used if there is no guarantee on the clock frequency ratios. Otherwise, a synchroniser may be replaced by the FIFO if one domain is known to be faster.

Table 4.1: Implemented CDC architectures with corresponding synchroniser types

	Architecture	push2pop	pop2push
1	Default	Handshake	Gray
2	Pop is faster	FIFO	Gray
3	Push is faster	Handshake	FIFO
4	CDC disabled	–	–

4.1 Implementation

The module was implemented at register-transfer level in SystemVerilog. Appendix A provides an overview of the structure of the module.

This section will further detail some implementation aspects which are critical to the CDC segmented buffer functionality. In particular, how segment counters are generated and how they are used for RAM access and segment status calculation.

4.1.1 Counter and Pointer Generation

The read and write counters, one for each virtual FIFO, must be generated according to the Gray subspace restrictions shown in figure 2.5. Afterwards, the virtual FIFO counters must be mapped into the corresponding physical RAM address. This section describes the implemented generation logic.

To place the counters in the corresponding Gray subspace, we must restrict counter start (ctr_start) and end (ctr_end) values according to equations 4.1 and 4.2. Note how these equations simplify if the segment depth is a power of two.

Virtual FIFO counter generation follows circular FIFO logic coupled with counter start and end restrictions, resulting in equation 4.3 which shows the forward counter value, i.e. the next value the counter should take after one RAM access. The write-side counter generation has additional conditional branching related to packet confirmation and cancellation; however, this extraneous logic is not included in this section.

$$ctr\_start = addr\_offset = 2^{\lceil log2(seg\_dp) \rceil} - seg\_dp \tag{4.1}$$

$$ctr\_end = 2 * seg\_dp + addr\_offset - 1 \tag{4.2}$$

$$ctr\_fwd(ctr) = \begin{cases} ctr + 1 & ctr \neq ctr\_end \\ ctr\_start & ctr = ctr\_end \end{cases} \tag{4.3}$$

The physical RAM was segmented according to section 1.1, defining segment address limits according to equations 4.4 and 4.5. Remapping the counter value into the physical RAM

address involves aligning the counter to RAM space and then adding the segment start address to the aligned counter (equation 4.10).

Aligning the counter to RAM space involves removing the wrap-status bit and, if the bit is not set, removing the initial address offset (equation 4.9). In essence, counters with set wrap-status bit are considered to be aligned, and counters with reset status bit are considered to be unaligned. Equations 4.6, 4.7 and 4.8 define the necessary wrap-status bit access and removal logic. Note how RAM alignment address offset compensation is not needed for segment depths that are powers of two.

$$ram\_seg\_start(i) = i * seg\_dp,\ i \in 0,...,(n\_segs-1) \tag{4.4}$$

$$ram\_seg\_end(i) = (i+1) * seg\_dp - 1,\ i \in 0,...,(n\_segs-1) \tag{4.5}$$

$$ctr\_wd = \lceil log2(seg\_dp) \rceil + 1 \tag{4.6}$$

$$msb(ctr) = ctr[ctr\_wd-1] \tag{4.7}$$

$$strip\_msb(ctr) = ctr[ctr\_wd-2:0] \tag{4.8}$$

$$align\_ctr(ctr) = strip\_msb(ctr) - \begin{cases} 0 & msb(ctr) = 1 \\ addr\_offset & msb(ctr) = 0 \end{cases} \tag{4.9}$$

$$ram\_ptr(seg, ctr) = align\_ctr(ctr) + ram\_seg\_start(seg) \tag{4.10}$$

Figure 4.1 shows these counter and pointer equations put in practice. With a segment depth of 6, virtual FIFO counter values are 4-bit wide, therefore allowing values from 0 to 15. To respect Gray restrictions on wrap-around, the counter values which would typically range from 0 to 11 must be shifted up two positions and count from $ctr\_start = 2$ to $ctr\_end = 13$. When aligning to RAM, in addition to stripping the MSb of the counter which indicates wrap-around state, we must match the counter values for both wrap-around states, which is done by shifting the $MSb = 0$ counter half upwards 2 positions, linking the counter value $ctr\_start = b0010$ to the counter value $ctr\_start + seg\_dp = b1000$. Afterwards, the virtual FIFO address is mapped to the physical segment 2 by adding $ram\_seg\_start(2)$ to the aligned counter.

4.1.2 Segment Status Calculation

Segment status calculation is implemented as described in section 2.2.4.1 coupled with the counter alignment logic of section 4.1.1. For this, the virtual FIFO push and pop counter

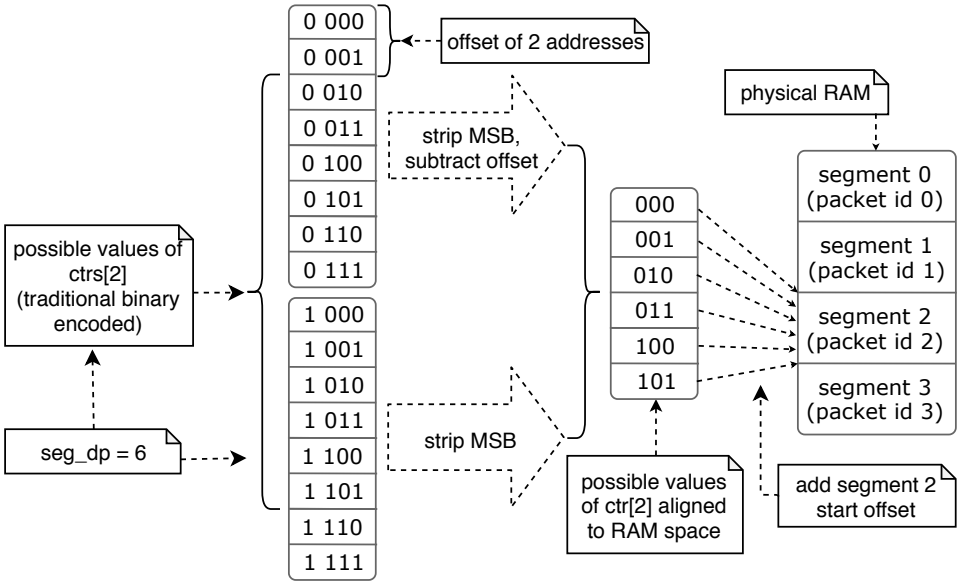


Figure 4.1: Flow of the virtual counter to physical pointer remap for segment 2, $seg\_dp = 6$

values ($push\_ctr$ and pop_ctr) must be used, each domain receiving the counter value of the other domain through the synchronisers.

Empty status occurs when both virtual FIFO counters, including the wrap-status bit, are equal. No counter alignment is necessary as empty status only occurs when both counters have the same wrap-status bit, so empty status is simply the direct comparison of both counters, as shown in equation 4.11.

Full status occurs when both aligned virtual FIFO counters are the equal (equation 4.12) and the wrap-status bit is different (equation 4.13), meaning that the write pointer has lapped the read pointer. Alignment is required because full status implies a comparison of counter values for counters with different wrap-status, leading to equation 4.14.

$$seg\_empty(seg) = \begin{cases} 1 & push\_ctr(seg) = pop\_ctr(seg) \\ 0 & otherwise \end{cases} \tag{4.11}$$

$$algn\_eq(seg) = \begin{cases} 1 & algn\_ctr(push\_ctr(seg)) = algn\_ctr(pop\_ctr(seg)) \\ 0 & otherwise \end{cases} \tag{4.12}$$

$$msb\_eq(seg) = \begin{cases} 1 & msb(push\_ctr(seg)) = msb(pop\_ctr(seg)) \\ 0 & otherwise \end{cases} \tag{4.13}$$

$$seg\_full(seg) = \begin{cases} 1 & algn\_eq(seg) = 1 \wedge msb\_eq(seg) = 0 \\ 0 & otherwise \end{cases} \tag{4.14}$$

4.2 Verification

This section describes how the implemented module was verified. We start by describing the main approach which consisted in the development of a dedicated testbench, followed by other verification methodology.

4.2.1 Dedicated Testbench

Verification of synchronous-side logic was verified through the implementation of a dedicated testbench. The testbench was extended to verify clock-domain crossing functionality through the generation of clock signals with randomised frequency changes and injection of CDC jitter in the fundamental synchronisation blocks. The testbench architecture follows the structure shown in figure 4.2.

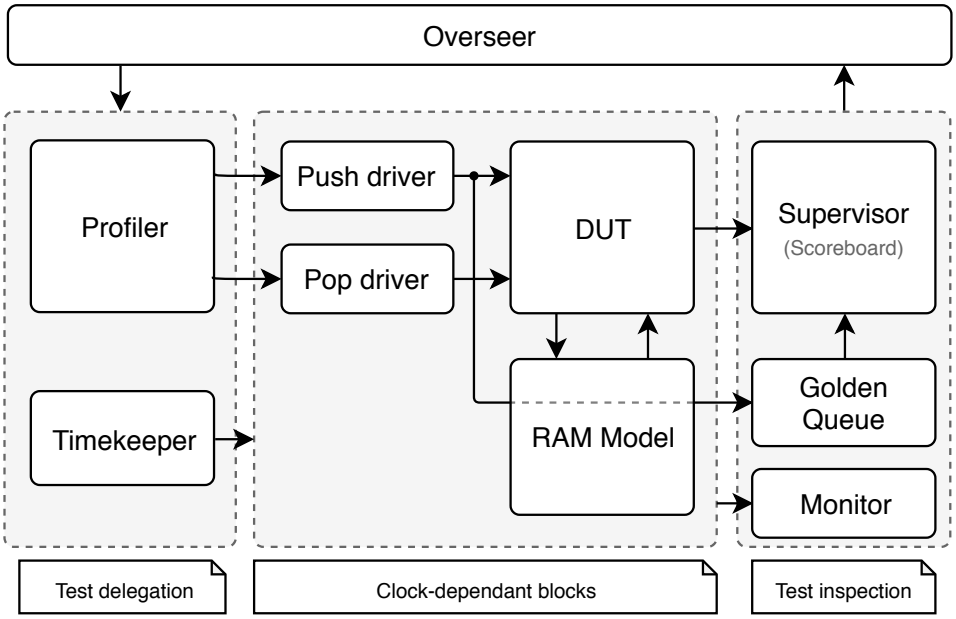


Figure 4.2: Block diagram of the implemented dedicated testbench

The testbench is comprised of four primary blocks characterised by their functionality:

- **Overseer:** Main test coordination block which iteratively commands the test delegation block to run tests and validates the results obtained by the test inspection block.
- **Test delegation:** Enables or disables the stimulus driving blocks. Composed by a profiler that generates randomised behavioural attributes that characterise the target activity of each driver. Composed by a timekeeper that generates reset and clock signals injected with randomised frequency changes.

- **Clock-dependant blocks:** All blocks that use the generated clock signals, including the design under test (DUT) coupled with a RAM model block with simulated internal latency. The DUT is driven by two stimulus generation blocks that follow the behavioural rules set by the test delegation. For push operations, the push driver generates completely random data.

- **Test inspection:** This block is responsible for collecting stimulus data and inspecting the resulting outputs for good behaviour. All data corresponding to kept elements is stored on a golden data queue which gets compared to the DUT data output when a pop command is asserted.

The test consists in the injection of push and pop commands according to a randomised behavioural profile which follows the set of attributes in table 4.2. This profile-based verification method is advantageous in terms of coverage as it allows increasing coverage metrics by running more test iterations and can be extended for corner-cases by forcing attributes.

Push behaviour is defined by sets of safety types according to table 4.3. Limiting the maximum complexity of the push behaviour depending on profile allows isolation and easier identification of problematic stimulus types.

Table 4.2: Testbench – Behavioural profile attributes

Attribute	Domain	Description
Activity	Both	Likelihood of asserting a request
Curiosity	Pop	Likelihood of reading without moving to next element
Granularity	Both	Tendency to burst in smaller sizes
Safety type	Push	Set of allowed push command types
Volatility	Push	Tendency to drop packets

Table 4.3: Testbench – Push safety types

Action	Non volatile	Safe volatile	Unsafe volatile	Description
Start	✓	✓	✓	Packet start
Push	✓	✓	✓	Packet push
Keep	✓	✓	✓	Packet confirm
Single push	✓	✓	✓	Single-element packet push and confirm
Drop	✗	✓	✓	Packet drop
Redrop	✗	✗	✓	Drop on dropped
Restart	✗	✗	✓	Start on started (implicit drop)
(Re)Drop-start	✗	✗	✓	(Re)Drop + Start
(Re)Drop-start-keep	✗	✗	✓	(Re)Drop + Single push

The testbench is capable of detecting both synchronous logic and CDC synchronisation problems as it performs full data integrity and status flag validity checks on both domains.

Bad synchronous logic would cause data corruption or erroneous status flag activation while problematic CDC paths would cause corruption of the remote counter values, therefore causing the same issues.

4.2.2 Other Verification Methods

Apart from the dedicated testbench designed to verify the module as a whole, we highlight two other verification stages the design underwent:

- **Segmented buffer drop-in replacement:** This test was performed in order to ensure full synchronous compatibility of the new segmented buffer (with integrated CDC) with an existing segmented buffer (single-clocked). It consists in instantiating the new segmented buffer alongside the old and assert output equivalence.

  The module was instantiated in complete PCIe subsystems in three instances. Full data equivalence was verified (with constant latency offsets of 0 to 2 clock cycles). Status flag equivalence was verified on an allow-pessimism basis (i.e., single-clocked SBC implies CDC SBC status flags, the opposite not being true) due to unavoidable CDC pessimism. In two of the three instances, full single-clocked SBC removal and replacement was performed. In the remaining instance, verification consisted in instantiating both modules side-by-side and comparing the outputs of both modules. This side-by-side approach was chosen due to the high difficulty of a full replacement for that particular instance, due to block interconnect complexity. Figure 4.3 shows the block diagram of the performed side-to-side verification.

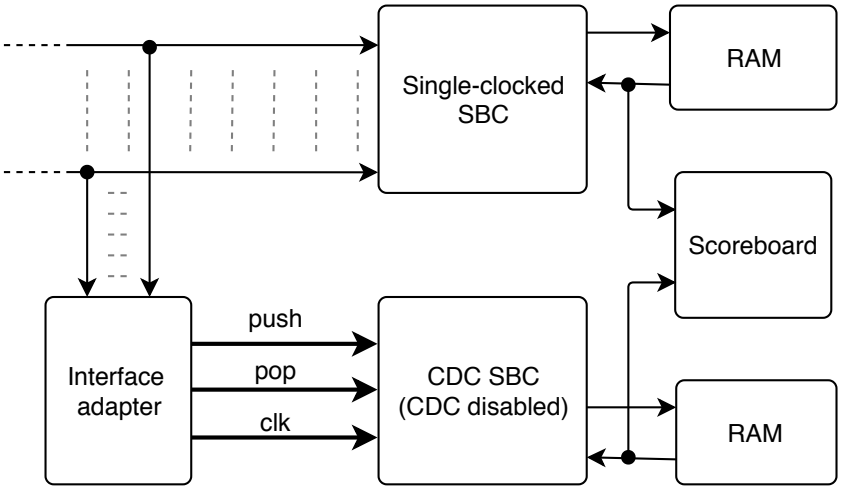


Figure 4.3: Block diagram of the performed side-by-side verification

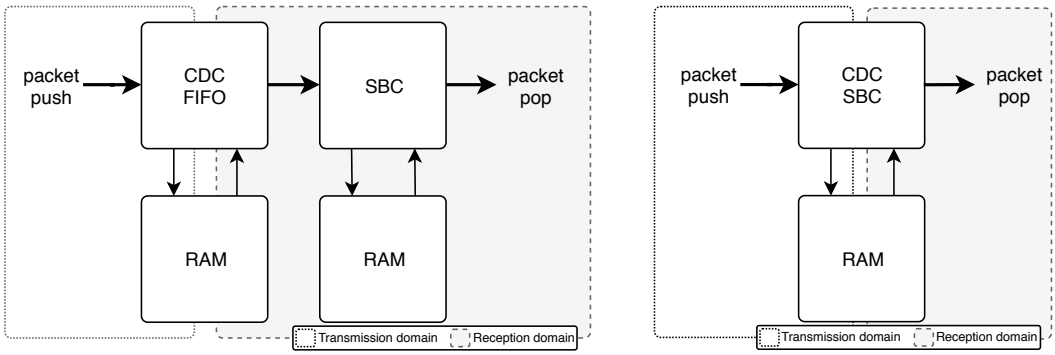
- **Automated checks:** The design was checked for CDC issues by RTL analysis tool Spyglass CDC [11]. The tool was configured to run the CDC abstract, CDC structural and clock reset integrity goals.

In order to use the tool, some configuration options must be passed first. The Gray encoded synchroniser requires specifying which signals are intended to be Gray encoded. This both allows the tool to constrain the input signals and allow them to pass through fundamental synchronisers without assuming incoherency in the resulting signal. The handshake synchroniser requires listing the REQ and ACK signals, allowing sampling triggers off those signals. Lastly, the FIFO synchroniser requires listing the shared memory, memory access pointers and FIFO input and output data buses.

4.3 Results and Discussion

Figure 4.4 shows the two compared CDC segmented buffer implementations. The comparison is between the existing solution, a single-clocked segmented buffer in series with a generic CDC FIFO (as described in chapter 1, shown in figure 4.4a), and the proposed solution consisting of the implemented segmented buffer module with integrated CDC (figure 4.4b). Additionally, for the proposed solution, the various CDC architectures shown in table 4.1 will be compared.

Standalone synthesis tests and functional performance tests will be carried out following the configurations of the scenarios A and C previously shown in table 3.2. These scenarios consist of a small configuration, with 4 segments of 18 depth each, and a large configuration, with 256 segments of 18 depth each. All tests will have the CDC FIFO parameterised to allow a maximum burst of one packet (18 elements).



(a) Existing solution

(b) Proposed solution

Figure 4.4: Existing vs proposed CDC packet transmission solutions

4.3.1 Synthesis Results and Area Evaluation

Post-synthesis area results were obtained for TSMC 16nm FinFET technology. Synthesis was performed in Design Compiler [10] and configured at Ultra High Effort with 30% clock uncertainty and 30% input delay on all ports.

Due to clock uncertainty, all paths must propagate in a maximum time of 70% of a nominal clock period and, additionally, lack of registering on the input and output lowers this by another 30% of a clock period. For example, a path that is neither registered at the input nor the output has a maximum allowed propagation delay of 10% of a clock cycle.

A significant problem of the existing solution is the second RAM used for the generic CDC FIFO. Not only is the final area cost worsened with the overhead of the additional RAM, but it also adds significant difficulties to the place-&-route phase. In particular, some RAM data widths are not supported by memory compilers, and one RAM may need to be split into multiple RAMs in order to support the data width.

After compilation, the extra RAM still presents layout difficulties [28]. The geometry of the block is dependant on its parameterisation: shallow RAMs with large data widths translate into long but thin rectangular layouts that may be hard to place. Furthermore, the RAM must be physically placed as close as possible to the stakeholder modules in order to minimise propagation delay, which further complicates placement.

Table 4.4 shows the synthesis area results for the scenarios. In terms of area, the implemented module is larger across the board even with CDC disabled. This can be explained by some factors:

- Removal of some segmented buffer controller optimisation logic that cannot be applied to CDC designs.
- Functional differences, e.g., the existing SBC does not support changing push packet ID without confirming or dropping the previous packet, while the proposed SBC does.
- In the proposed solution, CDC area scales logarithmically with the segmented buffer size, while in the existing solution it is constant.

Table 4.4: Results – Area (in thousands of gates)

Solution	Scenario A (4x18)	Scenario C (256x18)
Existing	17.8 (37% of RAM)	66.3 (0.54% of RAM)
Proposed (Default)	37.6 (78% of RAM)	409.6 (3.35% of RAM)
Proposed (Pop faster)	40.5 (84% of RAM)	238.2 (1.95% of RAM)
Proposed (Psh. faster)	42.3 (88% of RAM)	396.0 (3.24% of RAM)
Proposed (CDC off)	26.3 (54% of RAM)	176.4 (1.44% of RAM)

Regardless, when comparing the SBC size to the total RAM size, the SBC area remains reasonable and relative size decreases as segmented buffer size increases.

Table 4.5 shows synthesis success results at the 1 GHz working frequency goal for very large segmented buffer configurations. Two versions of the proposed solution were checked, one with support for pointer realignment as described in section 4.1.1 and another with this pointer realignment logic removed, meaning loss of support for the Gray encoded synchroniser.

Table 4.5: Results – Synthesis success for large configurations

Configuration	Existing	Proposed with realign	Proposed no realign
256x18	✓	✓	✓
1024x18	✓	✓	✓
2048x18	✗	✗	✓
4096x18	–	–	✓
8192x18	–	–	✗

The critical path of the proposed solution with support for pointer realignment is the generation of the RAM write pointer. This critical path consists of the conditional branching required by packet confirmation and cancellation followed by counter realignment logic and the addition of the RAM segment start offset. The removal of the counter realignment logic directly improves the propagation delay of this path.

4.3.2 Performance Results

Performance results were obtained based on functional simulations that measure the minimum amount of time required for the push side to write a given number of elements. The simulations start from reset and end when the push side has successfully pushed the required amount of elements.

A limitation of the existing implementation is performance degradation when the transmission clock domain is faster than the reception clock domain. The CDC FIFO is limited in size and can fill, causing transmission side halting while waiting for the reception side to process the pending elements. The proposed solution does not present this back-pressure limitation until the packet storage RAM is full.

Tables 4.6 and 4.7 show the push time simulation results for the comparison scenarios. In both tests, the burst was equal to the entire packet RAM depth. These tables represent the results for the best-case scenario for the proposed solution, as any future pushes would be limited by the reception domain, just like in the existing solution.

The obtained results are expected. Firstly, in the first column, both solutions perform similarly. This similarity is observed due to the push side being able to work at maximum speed on both solutions, as in the existing solution the pop domain can empty the CDC FIFO faster than the push domain can fill it.

On the second column, with frequencies reversed, the performance gain is significant with the new solution performing about twice as fast. The discrepancy between solutions is further steepened on the third column, where the proposed solution performs about 16 times better. These performance ratios are not coincidental; they are the ratio between the clock frequencies.

Table 4.6: Final tests – Time to complete burst – 4x18 – 72 element burst

Solution	4x18 Pop 1GHz Push 500MHz	4x18 Pop 500 MHz Push 1 GHz	4x18 Pop 62.5 MHz Push 1 GHz
Existing	150 *ns*	162 *ns*	1272 *ns*
Proposed (Default)	154 *ns*	78 *ns*	78 *ns*
Proposed (Pop faster)	152 *ns*	–	–
Proposed (Psh. faster)	–	79 *ns*	78 *ns*

Table 4.7: Final tests – Average pushes per *ns* – 256x18 – 4608 element burst

Solution	256x18 Pop 1GHz Push 500MHz	256x18 Pop 500 MHz Push 1 GHz	256x18 Pop 62.5 MHz Push 1 GHz
Existing	0.47	0.49	0.06
Proposed (Default)	0.47	0.95	0.95
Proposed (Pop faster)	0.47	–	–
Proposed (Psh. faster)	–	0.95	0.95

The performance difference between both solutions can be explored algebraically. The push domain is either limited by its clock or the reception domain clock. Let us formulate this burst performance comparison by assuming two things: that the storage RAM depth is larger than the CDC FIFO depth ($ram\_dp > fifo\_dp$), and that the push domain is working faster than the pop domain ($f\_push > f\_pop$). These assumptions can be mirrored in domains if the single-clocked SBC is placed in the pop domain.

In the existing solution, the push domain is limited by its clock until the synchronisation FIFO fills, thereafter being limited by the pop domain clock. This can be approximated by just assuming the FIFO will fill after $fifo\_dp$ pushes, which gets more accurate as the frequency ratio increases. Therefore, the number of successful pushes (n_pushes) vs the size of the burst attempt ($push\_clks$) in the existing solution can be described by equation 4.15.

Likewise, the proposed solution gets limited by the pop domain when the storage RAM fills, as seen in equation 4.16.

With this, we can trace three operating regions:

1. $push\_clks \in [0, fifo\_dp]$: Both solutions limited by the push clock
2. $push\_clks \in [fifo\_dp, ram\_dp]$: Existing solution now limited by the pop clock
3. $push\_clks \in [ram\_dp, \infty]$: Both solutions limited by the pop clock

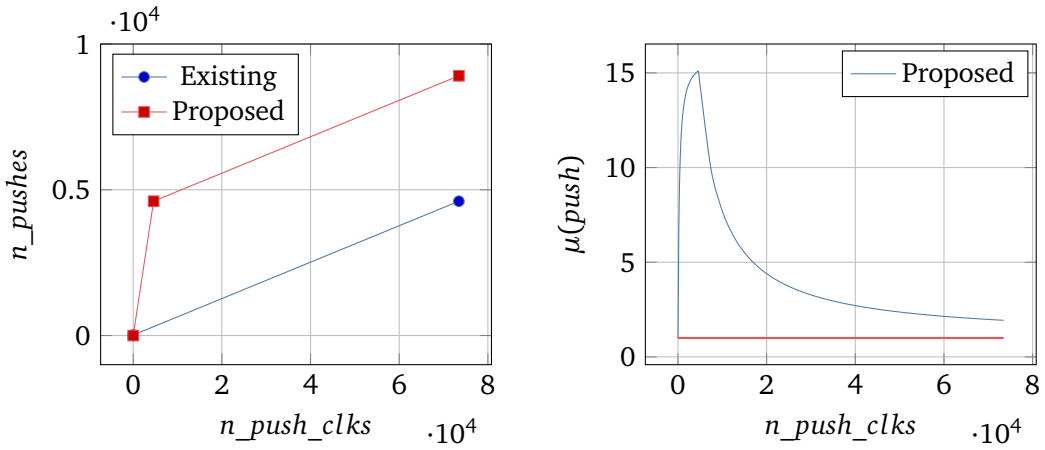
The burst performance gain of the proposed solution is obtained by dividing the number of successful pushes of both solutions in all operating regions, as described by equation 4.17. These equations are plotted for the rightmost column in figure 4.5. In figure 4.5a, n_pushes_1

is represented for the existing solution and n_pushes_2 for the proposed solution. The push efficiency of the proposed solution is plotted in figure 4.5b.

$$n\_pushes_1(push\_clks) = \begin{cases} push\_clks & push\_clks \leq fifo\_dp \\ fifo\_dp + \frac{f_{pop}}{f_{push}}(push\_clks - fifo\_dp) & otherwise \end{cases} \tag{4.15}$$

$$n\_pushes_2(push\_clks) = \begin{cases} push\_clks & push\_clks \leq ram\_dp \\ ram\_dp + \frac{f_{pop}}{f_{push}}(push\_clks - ram\_dp) & otherwise \end{cases} \tag{4.16}$$

$$\frac{n\_pushes_2}{n\_pushes_1}(push\_clks) = \begin{cases} 1 & push\_clks \leq fifo\_dp \\ \frac{push\_clks}{fifo\_dp + \frac{f_{pop}}{f_{push}}(push\_clks - fifo\_dp)} & fifo\_dp < push\_clks < ram\_dp \\ \frac{\left(\frac{f_{push}}{f_{pop}} - 1\right) \cdot fifo\_dp + push\_clks}{\left(\frac{f_{push}}{f_{pop}} - 1\right) \cdot ram\_dp + push\_clks} & otherwise \end{cases} \tag{4.17}$$



(a) Number of pushes

(b) Push efficiency

Figure 4.5: Final tests – Burst performance comparison, 1 GHz to 62.5 MHz, 256 segments

4.4 Summary

This chapter expands on the implementation of the proposed CDC SBC solution by formulating the virtual FIFO counter (equation 4.3) and RAM pointer (equation 4.10) generation logic required for desired segmented buffer functionality with support for Gray encoded synchronisation. Equations 4.11 and 4.14 formulate how the generated (and synchronised) virtual FIFO counters can be used to calculate segment full and segment empty status information. Section 4.2 aided the implementation sections by giving an overview of the used verification methodology.

The proposed CDC SBC implementation was compared against a generic implementation consisting of a single-clocked segmented buffer controller in series with a generic CDC FIFO. The proposed solution incurs in an additional area cost that scales logarithmically with segmented buffer size but provides a significant burst performance improvement that scales up to the domain clock ratios. For the generic implementation to reach this burst performance improvement, it would need another RAM the size of the packet storage RAM dedicated to synchronisation.

The proposed solution also provides other functional improvements, such as being able to provide segment status information directly to both clock domains. In the generic solution, this would require the addition of another synchroniser.

Chapter 5

Conclusions and Future Work

Chapter Outline

An ASIC module, consisting of a segmented buffer with integrated CDC, was designed and implemented. In terms of IC design flow progress, the work spanned the logic specification, logic design and early synthesis stages.

The performed work explored various concepts of clock domain crossing and data transmission. It presented an architecture designed to be resilient against the latency and jitter inherent to CDC, translating to a design where all single-clocked blocks show pessimistic behaviour until confirmation of non-worst-case status arrives through the synchronisation blocks. The work provides insight on typical synchronisation structures and their tradeoffs, their implementation and impact in physical metastability mitigation.

The implemented module feature set allows the usage of this segmented buffer in real PCIe devices. Aside from the development of a dedicated testbench, verification efforts included its instantiation alongside and, in some cases, full replacement of the previous segmented buffer solution in PCIe device configurations, although full integration on a PCIe subsystem requires changes outside of the scope of this work. Early synthesis results indicate successful synthesis at the 1 GHz goal for TSMC 16nm FinFET even for very large segmented buffers, up to 4096 segments. Due to this, the module is preliminarily viable for full integration in industry PCIe IP.

5.1 Review of Initial Questions

The questions posed in section 1.1 laid the groundwork for the goals of this dissertation by presenting a few key questions. Here, we would like to present a review of these questions and how they were explored.

- **Question 1.** How do we map data transactions between functional blocks into a simple structure that facilitates asynchronous processing?

  This question highlights how the module was intended to fit into the bigger picture of SoC interconnect. While not intended to be explored in-depth, it formulates the main functional aspects of the module. Figures 1.1 and 1.2 present the data structure, which is based on existing Synopsys designs.

  This data structure facilitates asynchronous processing through two main approaches. Firstly, packet confirmation or cancellation allows the transmitting domain to continuously write a data stream which does not have to be confirmed as valid immediately, allowing processing blocks to work in parallel with control blocks. Secondly, the assignment of a packet to a unique ID allows packets to be transmitted out of order and optionally re-ordered at reception. In the request/completion application shown in figure 1.2, this allows devices to send and receive request and completion data without requiring additional processing order control.

- **Question 2.** How do we map this data structure onto a physical memory?

  This question highlights another important aspect of the module as it is intended to control a single external RAM, which in turn serves as a shared platform for data storage.

  Mapping this data structure to RAM consists in assigning address regions to each data packet, as shown in figure 1.3. Similarly to question 1, this question was intended to build a base for the functional specification of the module and not intended to be studied in-depth. Some consequences of this RAM address mapping were further explored in sections 2.2.4.1 and 4.1.2, which shows how these RAM addresses can be used to calculate segment status flags, and sections 3.3.1 and 4.1.1, in which these RAM address regions must be compensated with an offset in order to allow segment depths that are not powers of two.

- **Question 3.** How do we allow this physical memory to be safely shared by two clock domains?

  This question provides the main point of exploration of this dissertation – the consequences of the consideration of this question span the entire length of the document. Chapter 2 served as a starting point, which revealed the critical problems of data sharing across clock domains. Chapters 3 and 4 detail the progress towards an architecture and module implementation that allow integration of CDC into the module. The goal of this CDC integration is to provide the domain-crossing conditions required for safe data sharing.

- **Question 4.** How do we do this while maximising performance?

  This question highlights a metric of quality that must be achieved through the development of the module. The question sets the goal to achieve the maximum module

performance while retaining acceptable area constrains. To achieve these restrictions, synchroniser performance was tested through RTL simulation as shown in section 3.5, and the full SBC implementation performance in section 4.3.2. Furthermore, the module must be designed to achieve high synthesis frequencies, meaning RAM address allocation and status flag calculation needs short combinational paths, thus studied in sections 2.2.4.1 and 4.1.1. Additionally, the architecture presented in section 3.2 had not only to be designed accounting for the latency and jitter of CDC but also to minimise latency to RAM access and to synchronisation across the push and pop sides. Chapter 4 presents an evaluation of the final module on this metric.

5.2 Future Work

The work presented in this thesis can be further explored and improved. We suggest the following routes:

- **Improved comparisons and benchmarks:** The performance evaluation for both the synchroniser tests (section 3.5.2) and the final module tests (section 4.3.2) were performed in standalone tests designed with this module in mind. This type of evaluation can result in a biased comparison and a better way to approach this evaluation would be to implement these changes in generic benchmarks (e.g., Gaussian Blur, Matrix Addition, Edge Detection, ...). Implementing generic benchmarks would require either finding benchmarks directly suited to testing CDC or adapting other benchmarks for this purpose.

- **Sharing the same RAM for packet data and synchronisation data:** The implemented module uses the RAM solely for the storage of packet data, with CDC data synchronisation handled externally in flip-flop-based logic. For very large segmented buffers this is not ideal as the flip-flop utilisation may grow to proportions where it would benefit from a dedicated RAM. An alternative to this is to store synchronisation data alongside packet data in the RAM. For example, the RAM data could be concatenated with an extra bit that indicates if there is more data available to read. When popping, the pop side could calculate “empty” status directly from this bit. Another option is having a dedicated memory region where all synchronisation data is stored. The main difficulty behind this is handling RAM access, as it would require developing a system that switches from packet read and write to pointer synchronisation activity. Furthermore, a small additional synchronisation mechanism would be required to avoid simultaneous accesses to synchronisation memory regions.

- **Dynamic segment sizes:** A significant limitation of the proposed design are the hard boundaries in segment size. Segment depth is static and defined through parameterisation to be equal to the MTU, which results in relatively inefficient usage of the RAM

space as in some cases the segments could remain empty almost all of the time, awaiting packet arrival.

RAM utilisation efficiency can be improved through various approaches in exploration of dynamic memory architectures. One approach is reworking RAM address abstraction to allow flexible segment size, allowing a reduction of the RAM size. However, flexible segment sizes introduce other conceptual issues. If the segmented buffer controller allows one segment to take the entire RAM space, packets may be completely blocked from entry, causing propagation of back-pressure through the system. The implementation of a dynamic segment depth solution provides an opportunity for the study and development of a scheduler or arbiter that manages quality of service.

Appendix A

Implementation: SysML to SystemVerilog

The module implementation followed Synopsys methodology consisting of automatic RTL generation from SysML based on work by Oliveira et al. [24]. This appendix includes the relevant SysML Enterprise Architect [30] structural diagrams.

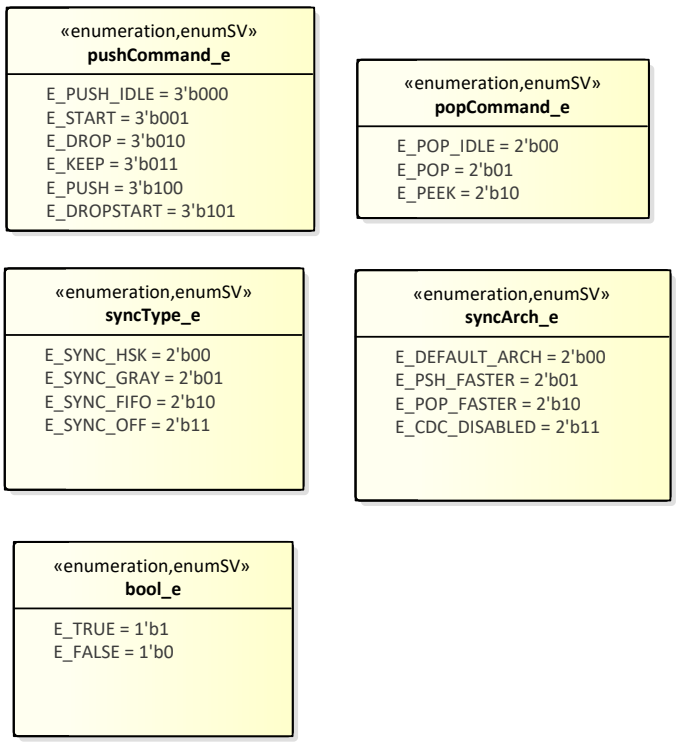


Figure A.1: SysML SystemVerilog enumeration definitions

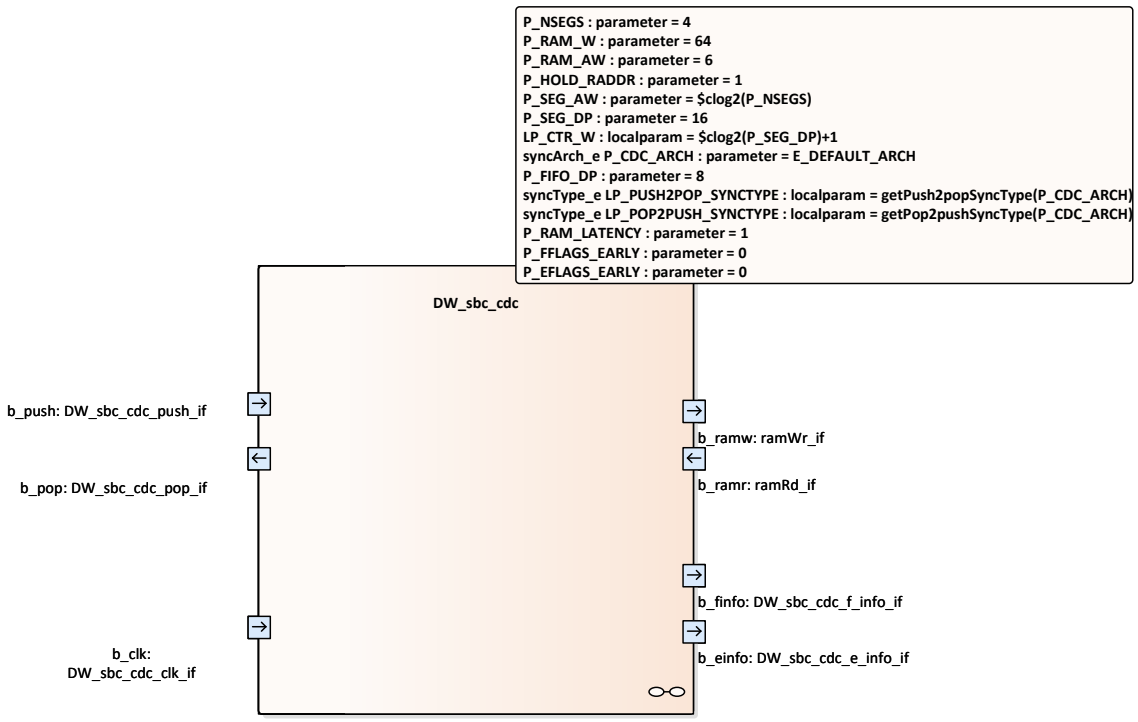


Figure A.2: SysML SystemVerilog top-level block definition

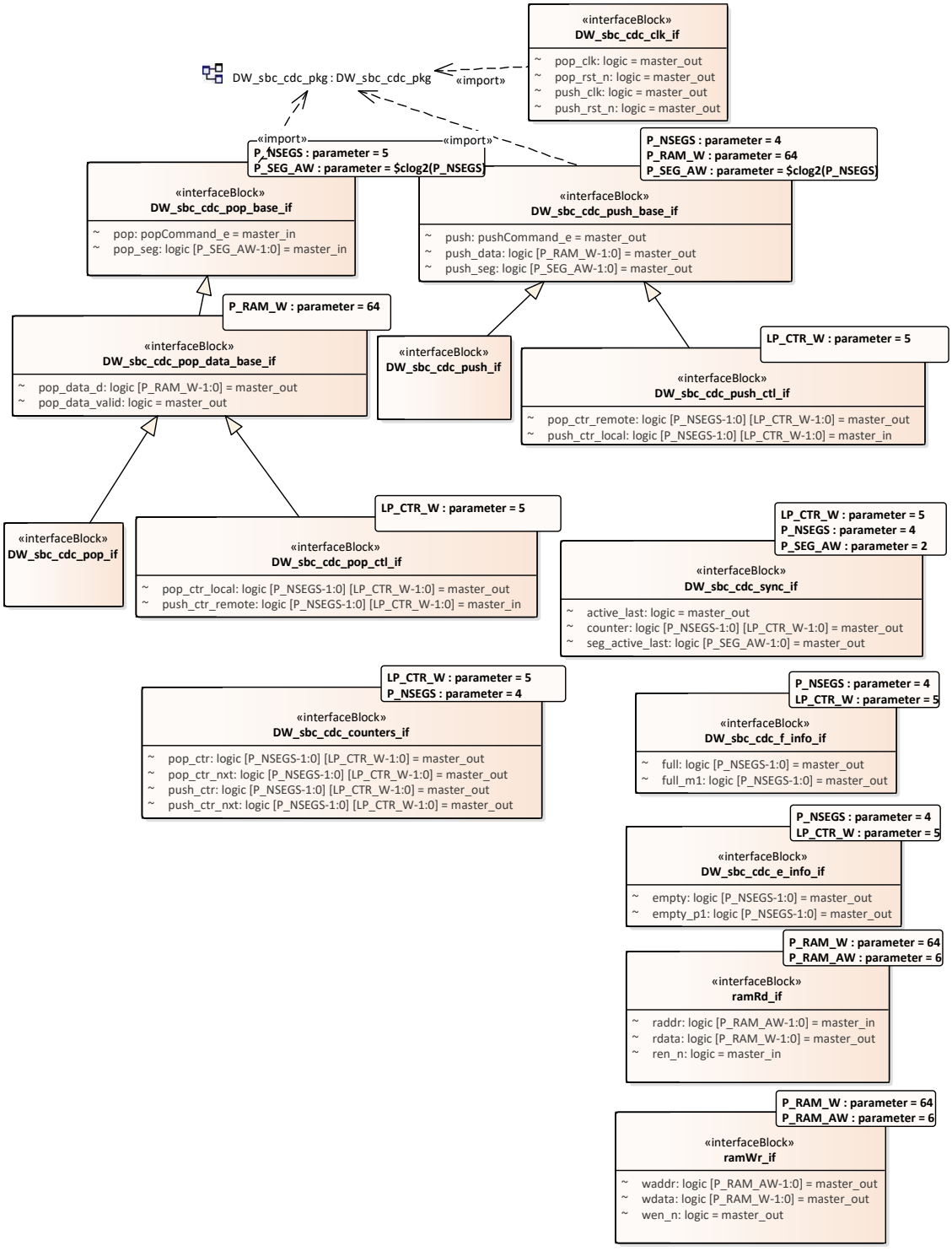


Figure A.3: SysML SystemVerilog interface hierarchy

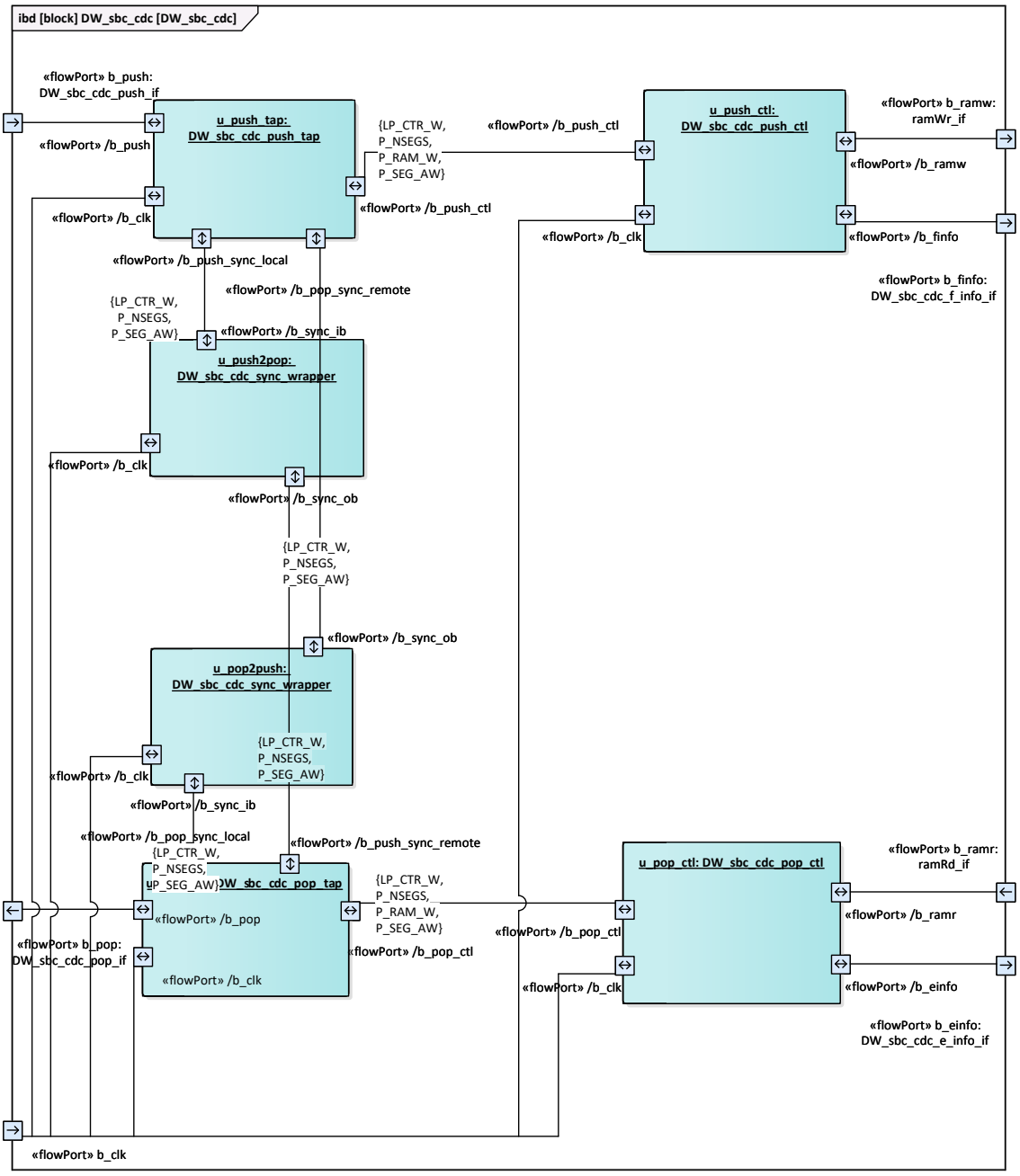


Figure A.4: SysML SystemVerilog top-level internal block diagram

References

[1] Flavio Giovanni Bonomi, Albert Gordon Greenberg, and Jennifer Lynn Rexford. Method for leaky bucket traffic shaping using fair queueing collision arbitration, November 3 1998. US Patent 5,831,971.

[2] Milena Butto, Elisa Cavallero, and Alberto Tonietti. Effectiveness of the 'leaky bucket' policing mechanism in ATM networks. *IEEE Journal on selected areas in communications*, 9(3):335–342, April 1991.

[3] Doris Chen, Deshanand Singh, Jeffrey Chromczak, David Lewis, Ryan Fung, David Neto, and Vaughn Betz. A comprehensive approach to modeling, characterizing and optimizing for metastability in FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA '10*, pages 167–176. ACM Press, 2010.

[4] Clifford E. Cummings. Simulation and Synthesis Techniques for Asynchronous FIFO Design. In *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*, March 2002. Section TB2, 2nd paper. Available: http://www.sunburst-design.com/papers [Accessed: 2019-06-18].

[5] Clifford E. Cummings. Clock Domain Crossing (CDC) Design & Verification Techniques Using SystemVerilog. In *SNUG 2008 (Synopsys Users Group Conference, Boston, MA, 2008) User Papers*, September 2008. Available: http://www.sunburst-design.com/papers [Accessed: 2019-06-18].

[6] Tejas Dave, Amit Jain, and Divyanshu Jain. Synchronizer techniques for multi-clock domain SoCs & FPGAs. *EDN Network*, September 2014. Available: https://www.edn.com/electronics-blogs/day-in-the-life-of-a-chip-designer/4435339/Synchronizer-techniques-for-multi-clock-domain-SoCs [Accessed: 2019-06-18].

[7] T.J. Gabara, G.J. Cyr, and C.E. Stroud. Metastability of CMOS master/slave flip-flops. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(10):734–740, 1992.

[8] Paul J Giacobbe and Robert P Ryan. Frequency independent asynchronous clock crossing FIFO, August 1 2000. US Patent 6,098,139.

[9] R. Ginosar. Metastability and synchronizers: A tutorial. *IEEE Design & Test of Computers*, 28(5):23–35, September 2011.

[10] Synopsys Inc. Design Compiler: RTL Synthesis. Web page. Available: https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html [Accessed: 2019-06-18].

[11] Synopsys Inc. Spyglass Products. Web page. Available: https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html [Accessed: 2019-06-18].

[12] T.A. Jackson and A. Albicki. Analysis of metastable operation in D latches. *IEEE Transactions on Circuits and Systems*, 36(11):1392–1404, 1989.

[13] Ian W. Jones, Suwen Yang, and Mark Greenstreet. Synchronizer behavior and analysis. In *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 117–126. IEEE, May 2009.

[14] Tsachy Kapschitz and Ran Ginosar. Formal verification of synchronizers. In *Lecture Notes in Computer Science*, pages 359–362. Springer Berlin Heidelberg, 2005.

[15] Lindsay Kleeman and Antonio Cantoni. Metastable behavior in digital systems. *IEEE Design & Test of Computers*, 4(6):4–19, 1987.

[16] Lindsay Kleeman and Antonio Cantoni. On the unavoidability of metastable behavior in digital systems. *IEEE Transactions on Computers*, 100(1):109–112, 1987.

[17] Chris Kwok, Vijay Gupta, and Tai Ly. Using assertion-based verification to verify clock domain crossing signals. In *Proc. of Design and Verification Conf.*, pages 654–659. DVCon, February 2003.

[18] Ka-Kei Kwok, Bing Li, Tai An Ly, and Rojer Raji Sabbagh. Formal verification of clock domain crossings, September 18 2012. US Patent 8,271,918.

[19] Bing Li and Chris Ka-Kei Kwok. Automatic formal verification of clock domain crossing signals. In *2009 Asia and South Pacific Design Automation Conference*, pages 654–659. IEEE, January 2009.

[20] Mark Litterick et al. Pragmatic Simulation-Based Verification of Clock Domain Crossing Signals and Jitter using SystemVerilog Assertions. *Proceedings of DVCon*, 6, 2006.

[21] Li Luo, Hongjun He, Qiang Dou, and Weixia Xu. Design and verification of multi-clock domain synchronizers. In *2010 International Conference on Intelligent System Design and Engineering Application*, volume 1, pages 544–547. IEEE, October 2010.

[22] Cayla McGinnis. PCI-SIG® Fast Tracks Evolution to 32GT/s with PCI Express 5.0 Architecture, June 2017. News Release. Available: https://www.businesswire.com/news/home/20170607005351/en/PCI-SIG&#xAE-Fast-Tracks-Evolution-32GTs-PCI-Express [Accessed: 2019-06-18].

[23] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965. 38(8):114–117, April 1965.

[24] Manuel Oliveira. A SysML-based Design Flow for Digital VLSI Circuits. Master's thesis, FEUP, Porto, 2015.

[25] Clemenz Lenard Portmann. *Characterization and reduction of metastability errors in CMOS interface circuits*. PhD thesis, Stanford University, 1995.

[26] Shaker Sarwary and Saurabh Verma. Critical clock-domain-crossing bugs. *Electron. Des. Strategy News*, 53(7):55–64, April 2008.

[27] N. Sharif, N. Ramzan, F. K. Lodhi, O. Hasan, and S. R. Hasan. Quantitative analysis of state-of-the-art synchronizers: Clock domain crossing perspective. In *2011 7th International Conference on Emerging Technologies*, pages 1–6. IEEE, September 2011.

[28] M. J. S. Smith. *Application-Specific Integrated Circuits*, chapter 16, pages 982–1042. Addison-Wesley, 1997.

[29] Cadence Design Systems. "Clock Domain Crossing: Closing the loop on clock domain functional implementation problems". White paper, 2004.

[30] Sparx Systems. Full lifecycle modelling for business, software and systems. Web page. Available: https://sparxsystems.com/products/ea/index.html [Accessed: 2019-06-18].

[31] Saurabh Verma and Ashima S Dabare. Understanding clock domain crossing issues. *Atrenta*, 2007. Available: http://www.eetimes.com/design/edadesign/4018520/Understanding-Clock-Domain-Crossing-Issues [Accessed: 2019-06-18].

[32] Jun Zhou. *Design and Measurement of Synchronizers*. PhD thesis, Newcastle University, November 2008.